VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Editor Petriho sítí
Petri Net Editor

2012

Pavel Kučera

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Pavel Kučera**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Editor Petriho sítí
Petri Net Editor

Zásady pro vypracování:

Cílem práce je vytvořit editor Petriho sítí na platformě Eclipse RCP, který bude umožňovat vytvářet bloky podsítí a ty postupně spojovat. Zadání práce je propojeno se zadáním Simulátor Petriho sítí.
Editor by měl umožňovat následující:
1. Vytvářet Petriho sítě v grafickém editoru.
2. Umožňovat vytvářet bloky Petriho sítí, které bude posléze možno vložit do celé Petriho sítě. Pro každý blok se určí množina vstupních a výstupních míst.
3. Editor umožní vytvořit několik grafických reprezentací jednoho místa v síti, aby bylo možno vyhnout se zbytečnému křížení hran.
4. Editovat parametry jednotlivých elementů Petriho sítě (vlastnosti míst, přechodů, vlastnosti jednotlivých tokenů).
5. K jednotlivým přechodům bude možno definovat skripty, které budou ovlivňovat průběh simulace.

Práce bude rovněž obsahovat přehled práce s platformou RCP a především projekt Graphical Modeling Project (GMP).

Seznam doporučené odborné literatury:

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Gang of Four): Návrh programů pomocí vzorů. Grada. Praha 2003. ISBN 8024703025
Eclipse [online]. 2011 [cit. 2011-06-04]. Dostupné z WWW: <http://www.eclipse.org/>.

Dále podle pokynů vedoucího práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. David Ježek, Ph.D.**

Datum zadání: 18.11.2011

Datum odevzdání: 04.05.2012

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení studenta

Prohlašuji, že jsem tuto bakalářskou/diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Prostřednictvím této práce bych rád poděkoval Ing. Davidu Ježkovi, Ph.D. za vedení a odbornou pomoc v průběhu vývoje této práce.

4.5.2012
.....
datum

Kuřera
.....
podpis

Abstrakt

Když jsem si vybral jako téma své diplomové práce Editor Petriho sítí, pomyslel jsem si, že to je téma přesně pro mě. Programování mě baví a Petriho sítě mi přišly jako velice zajímavé téma. Jenomže v okamžiku, kdy jsem se začal seznamovat s GMF technologií, pomocí níž jsem měl výsledku dosáhnout, začal jsem svého rozhodnutí trochu litovat. Tato technologie pro mě byla novinkou, dokumentace také nebyla příliš obsáhlá, proto jsem se začal trochu ztrácet. Ale nějaký čas studia GMF technologie začal přinášet ovoce. A když už jsem napsal první funkční kód, věděl jsem, že mám vyhráno. S přibývajícími zkušenostmi aplikace pomalu rostla a začala dostávat výslednou podobu. Když jsem se však blížil k závěru, byl také i čas neúprosný a já musel implementaci náhle ukončit, aniž bych splnil všechny body zadání. Ale i tak je aplikace ve funkčním stavu a připravená na další možná rozšíření.

Klíčová slova: Petriho síť, editor, GMF, podnikový proces, diagram tříd

Abstract

When I have chosen Petri net editor as a topic of my dissertation, I believed, it's the rigt topic for me. I like programming and Petri nets looked like very intesting topic. But at the moment, when I have begun to learn about GMF technology, throuhg the use of it I wanted to reach some results, I began to regret of my decision. This technology was completely new for me and the documentation wasn't so extensive, that's why I began lose myself in the topic. But after some time of studying GMF technology, situation became better. And when I have written my first usefull code I knowed, I did it. With more experiences my aplication gerw up and begun to have resulting form. But because deadline was very close yet I don't have so many time and I have to suddenly stopped the implementation without fulfillment of all articles of my dissertation. Despite of this problem is aplication functional and ready for some other possible extensions.

Key Words: Petri Net, editor, GMF, business process, class diagram

Seznam použitých symbolů a zkratek

CPN [Coloured Petri Net]	- barevná Petriho síť
EMF [Eclipse modeling framework]	- framework pro tvorbu java aplikací založených na strukturovaném modelu
GEF [Graphical editor framework]	- framework pro tvorbu editorů
GMF [Graphical Modeling Framework]	- kombinace EMF a GEF
GUI [Graphical User Interface]	- grafické uživatelské rozhraní
PN [Petri Net]	- Petriho síť
UML [Unified Modeling Language]	- jazyk pro vizualizaci, specifikaci a navrhování programových systémů

1 Obsah

2 Úvod

Tato práce je závěrečnou zprávou řešeného projektu na téma Editor Petriho sítí. V počáteční fázi vývoje vnikla jednoduchá analýza, která dala základ vývoji aplikace. Před zahájením vývoje jsem musel správně nastavit prostředí a seznámit se s technologií GMF. Samotný vznik projektu započal vytvořením diagramu reprezentujícího objektový model aplikace, dále vytvořením grafického modelu a modelu nástrojů a nakonec jejich skloubením, čímž jsem získal model mapování. Dále jsem provedl transformaci nad všemi modely a získal tak model, ze kterého jsem vygeneroval výsledný kód. Musel jsem ještě přizpůsobit editor potřebám zadání diplomové práce a nakonec otestovat.

3 Editor Petriho sítí

3.1 Cíle práce

Cílem diplomové práce je vytvoření grafického editoru, umožňujícího vytváření struktury PN (Petriho sítí) a nastavení vlastností a vztahů PN systému. Další funkcí výsledné aplikace by měla být také simulace vytvořené PN, která je předmětem diplomové práce „Simulátor Petriho sítí“. Celá aplikace je přizpůsobena především pro návrh podnikových procesů, proto musí být navržena jako barevná PN.

Editor umožňuje sestavení PN jako celku, případně po jednotlivých blocích (podprocesech), které vzájemným propojením tvoří výslednou PN. Podproces reprezentuje opět PN, která navazuje na definované vstupy a výstupy tvořící rozhraní mezi sítí a podprocesem. Každá PN je tvořena pomocí míst, přechodů a hran. Každé místo obsahuje atributy jako název a poznámky a také umožňuje definovat kapacitu místa, to znamená skupiny objektů, které se mohou v místě vyskytovat a jejich maximální počet v místě. Přechody mají také svůj název a poznámky, dále je zde možné psát skripty, které určují, jak se daná PN bude chovat při simulaci a vytvářet scénáře, jež definují změny objektů při provedení přechodu. Hrany slouží k propojení jednotlivých elementů sítě, kdy je u každé hrany definován zdrojový a cílový element. Každá hrana definuje také svou násobnost, tedy seznam skupin a jejich počet, který propustí. Dále si hrany uchovávají seznam skriptů, které lze na dané hraně aplikovat. V neposlední řadě editor umožňuje vytvářet a mazat již mnohokrát zmiňované skupiny objektů a samozřejmě samotné objekty. Ty mohou být buď aktivní, nebo pasivní. Aktivní objekty reprezentují většinou stakeholdery daného procesu, kdežto pasivní označují různé artefakty. Objekty si uchovávají svůj počet, případné poznámku a také skupinu, kterou reprezentují.

3.2 Charakteristika současného stavu řešené problematiky

V současné době je aplikace tvořena několika pluginy, které jsou jednoduše spustitelné ve vývojovém prostředí Eclipse. Pro práci v aplikaci je pro uživatele připraveno příjemné a přehledné grafické uživatelské rozhraní, díky němuž celkem rychle a intuitivně pochopí, jak se s aplikací pracuje. Avšak drobnou nevýhodou je, že uživatel nevidí některé podstatné vlastnosti různých elementů (např. skupinu objektu, násobnost hran, …). Tuto možnost mu poskytne až otevření příslušného dialogového okna dvojklikem na daný element, případně označení daného elementu a následné vyhledání potřených informací v okně vlastností. Dalším podstatným nedostatkem je také nesplnění jednoho z bodů zadání diplomové práce, který požaduje možnost vytvoření několika grafických reprezentací jednoho místa v síti. Tento bod zadání je splněn jen

z části. Modelový objekt reprezentující místa v síti sice poskytuje možnost vytváření asociací mezi jednotlivými místy sítě, toto lze učinit pouze uživatelsky, programově tato problematika není vyřešena. Dalším nedostatkem celé aplikace je nepřítomnost některých požadovaných atributů u aktivních i pasivních objektů. Těmito atributy se rozumí například zkušenost aktivního objektu, kvalita pasivního objektu apod. Za poslední známý nedostatek považuji to, že v aplikaci není ošetřené propojování jednotlivých elementů hranami. Např. pravidla PN neumožňují vzájemné propojení dvou míst hranou, což v této aplikaci není sebemenší problém.

3.3 Specifikace etap řešení

Jelikož technologie GMF pro mě byla novinkou, prvním důležitým cílem bylo pochopení této problematiky její použití v praxi. Toto však s sebou přineslo další komplikace v podobě problémů při nasazení technologie.

Po zvládnutí počátečních neúspěchů a komplikací a také nastudování nové technologie přišla na řadu analýza diplomové práce. Výsledkem byl ne příliš obsáhlý dokument, jež zachycoval nejdůležitější body zadání a první návrhy aplikace.

Tvorba samotné aplikace začíná vznikem metamodelu. Když jsem měl vytvořený metamodel, bylo potřeba z něj vygenerovat příslušné java třídy.

Dalším krokem bylo vytvoření grafické reprezentace prvků metamodelu. V této části jsem definoval v první řadě vzhled jednotlivých prvků diagramu, popřípadě kontextové menu apod.

Následovalo nastavení palety nástrojů pro vytváření jednotlivých elementů diagramu. Určoval jsem elementy a jejich nadřazené skupiny, do kterých spadají.

Spojením předchozích dvou částí jsem získal model, který definuje vzájemné vztahy mezi metamodelem a jeho grafickou reprezentací.

Dostal jsem se k poslednímu kroku postupného generování aplikace. Provedl jsem transformaci nad již vytvořenými modely (metamodel, grafický model a model nástrojů) a získal tak výsledný model editoru. Ten mi nabídl poslední možnost poupravit některé vlastnosti metamodelu. Nakonec jsem vygeneroval jednotlivé třídy výsledného editoru a tím získal základ celé aplikace.

Již jsem měl funkční kostru editoru a mohl jsem si jí pomalu formovat k obrazu svému. Začal jsem vlastním uspořádáním elementů v compartment procesu. Výrazem compartment je označována část určitého elementu, ve které se mohou vyskytovat jiné elementy. Proces (podproces) takové obsahuje 2 compartment, a to vstupní a výstupní. Defaultně se však při vytváření elementů v compartment elementy skládají jeden na druhý a v jejich původní

velikosti. Proto bylo třeba elementy uspořádat a upravit velikost tak, aby působily co nejpřehledněji a nezabíraly zbytečně velkou plochu palety editoru.

Dalším výrazným zákrokem do aplikace byla realizace dialogových oken jednotlivých elementů, které slouží pro editaci jejich jednotlivých atributů.

3.4 Postup zpracování řešené problematiky

3.4.1 Přizpůsobení pro podnikové procesy

Pojmem proces se rozumí posloupnost aktivit směřujících k nějakému cíli. Podnikové procesy jsou potom procesy v podniku, které ve svém důsledku nakonec spějí k co nejvyššímu zisku. Petriho sítě jsou ideálním nástrojem pro modelování a simulaci procesů. Podnikový proces je tvořen:

- **Aktivitami**, které konzumují a vytvářejí objekty
- **Místy**, které přijímají definované typy objektů
- **Objekty** (aktivní, pasivní)

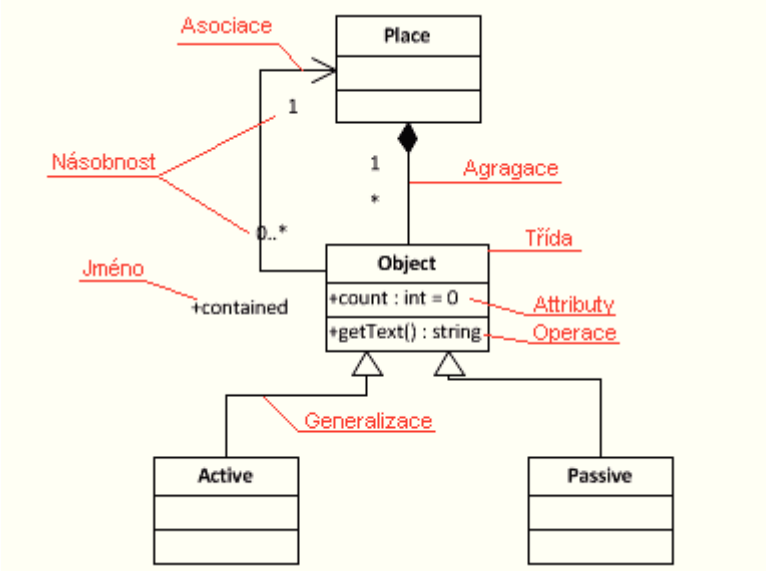
Podobnost s PN je zde velice vysoká. Problém však nastává v okamžiku, kdy hovořím o typech objektů. Klasické PN typy tokenů nerozlišují, proto vznikly Barevné Petriho sítě (CPN), které tento požadavek splňují. Pokud chci tedy vytvořit editor, jenž má sloužit k modelování podnikových procesů, vytvářím v podstatě editor CPN, jehož místa reprezentují místa procesu, přechody reprezentují aktivity, tokeny jsou jednotlivé objekty procesu a jejich typ rozliším barvou. Tedy v CPN se typ označuje barvou, ale protože barva mi o objektu nic moc nevypoví, v aplikaci člením objekty i do skupin.

3.4.2 Základy UML

UML je grafický jazyk pro vizualizaci, specifikaci a navrhování programových systémů. Diagramy používané v UML jsou rozčleněny do 3 skupin:

- Strukturní diagramy
- Diagramy chování
- Diagramy interakce

Já se budu zabývat pouze strukturním diagramem potřebným pro tvorbu GMF aplikací a tím je diagram tříd. Tento diagram slouží k popisu objektů a statických vztahů mezi nimi.



Obrázek 1: Diagram tříd

Na obrázku jsou popsány jednotlivé prvky diagramu tříd. Většina z nich je vysvětlena v

Tabulka 2. Vztahy mezi objekty (Třídami) mohou být trojího typu:

- Asociace
  - Je to spojení mezi objekty, to znamená, že jeden objekt má vytvořený odkaz na druhý a násobnost jejich asociace určuje, kolik odkazů může který objekt mít. Asociace z Obrázek 1 tedy značí, že místo může být odkazováno na libovolný počet objektů, ale objekt musí mít odkaz na právě 1 místo.

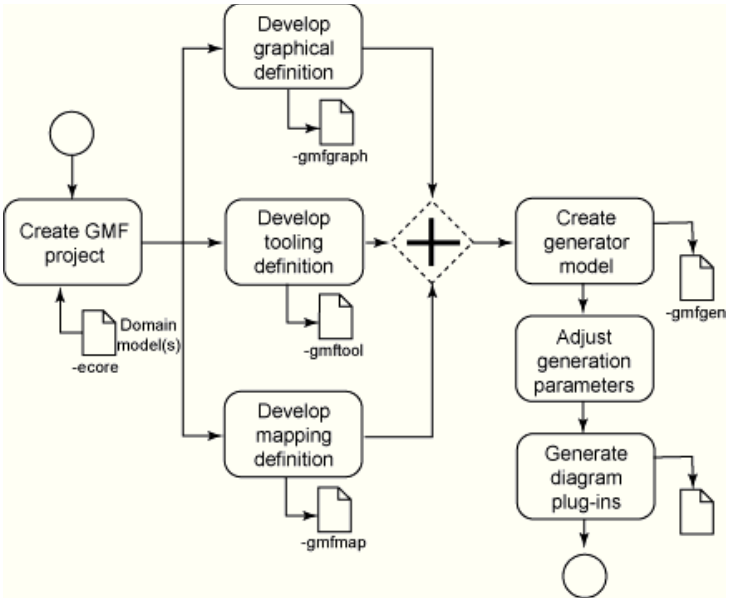
- Agregace
  - Popisuje vztah mezi celkem a jeho částmi. Zohledníme-li i příslušné násobnosti, můžeme pak říct, že místo obsahuje libovolný počet objektů, ale objekt musí patřit pouze jednomu místu.

- Generalizace
  - Tento vztah reprezentuje dědičnost mezi objekty, pokud tedy objekt obsahuje atribut „count“ a operaci „getText()“, tak aktivní a pasivní objekty je obsahují také.

3.4.3 Tvorba GMF aplikací

Eclipse GMF (Graphical modeling framework) je sada zásuvných modulů do nástroje Eclipse, umožňujících vytváření metamodelů a modelovacích editorů. Při práci s GMF je využívána řada dalších modulů. Mezi ty nejdůležitější patří EMF a GEF. GMF poskytuje celou paletu možností a funkcí, které rozšiřují možnosti metamodelování. Jsou jimi např. validování modelů, tvorba kontextových menu nebo generování kódu.

Samotný vývoj GMF aplikací začíná vytvořením metamodelu, následuje definice vzhledu editoru, panelu nástrojů a vztahů mezi jednotlivými elementy a končí vygenerováním příslušných tříd výsledné aplikace. Obrázek 2 názorně popisuje celý vývojový proces.



Obrázek 2: Proces vývoje GMF aplikace

Po vytvoření GMF projektu je nutno přidat soubor s koncovkou *.ecore* definující metamodel. Ten lze buď vytvořit, vygenerovat nebo importovat již existující. Ze souboru *.ecore* obsahujícího model odvodím následující soubory:

- *.genmodel* – slouží k vygenerování tříd modelu, popř. dalších volitelných tříd
- *.gmfgraph* – grafická reprezentace prvků metamodelu
- *.gmftool* – nastavení chování generovaného editoru jako kontextové menu a obsah palet
- *.gmfmap* – reprezentuje propojení předchozích souborů, jako například který obrázek náleží jakému prvku modelu

Transformací těchto 3 souborů získám soubor s koncovkou *.gmfgen*, z něhož jsou na závěr vygenerovány příslušné java třídy výsledného editoru. Po dokončení celého procesu bych měl mít vygenerovány tyto projekty:

- <název> - třídy jednotlivých objektů modelu
- <název>*.diagram* - práce s elementy editoru
- <název>*.edit* - pomocné třídy
- <název>*.editor* - třídy definující např. kontextová menu editoru
- <název>*.test* - testovací třídy

Spuštěním těchto pluginů v prostředí Eclipse získám editor se základními funkcemi, jako jsou vytváření jednotlivých elementů diagramu, vzájemné propojování hranami, vytváření podprocesů a základní nastavení vlastností.

3.4.4 Příprava vývojového prostředí

Jedním z dobrých předpokladů pro vývoj kvalitní aplikace je bezpochyby dobře fungující pracovní prostředí. Nezdá se to být nic složitého nainstalovat pár nástrojů pro práci, avšak opak může být pravdou, proto zde uvedu podrobný postup, jak jsem si připravil funkční prostředí pro vývoj GMF aplikací v Eclipse na platformě Windows.

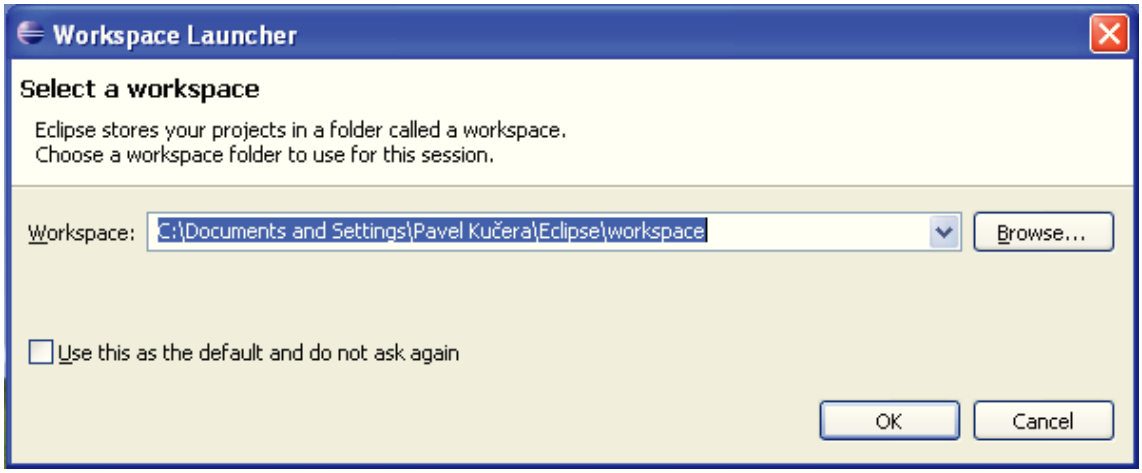
Software potřebný pro instalaci:

- Windows XP Professional (x86)
- Eclipse Modeling Tools (Verze: Indigo Service Release 2)
- Java SE Runtime Environment 7u3 (JRE 7 Update 3)

Samozřejmě doba jde kupředu a budou vznikat nové verze, proto uvádím především ty ověřené a je na každém, zda ve verzích povýší či se bude držet těch výše uvedených.

Instalací operačního systému se zde nebudu zabývat, to jistě každý zvládne sám. Hlavní věc, kterou se budu zabývat, je prostředí Eclipse. Toto je zdarma ke stažení na stránkách http://www.eclipse.org/ . Doporučuji stáhnout nástroj Eclipse Modeling Tools, který je přímo přizpůsobený pro modelování v Javě. Tento však potřebuje ke svému běhu také nainstalovanou Javu, a to buď JDK, nebo alespoň JRE. Já jsem zvolil JRE ve verzi 7.0.30 a vše funguje jak má. Java je také volně ke stažení na stránkách společnosti Oracle, tedy na http://www.oracle.com .

Po prvním spuštění Eclipse jsem byl dotázán na umístění projektů (Workspace).

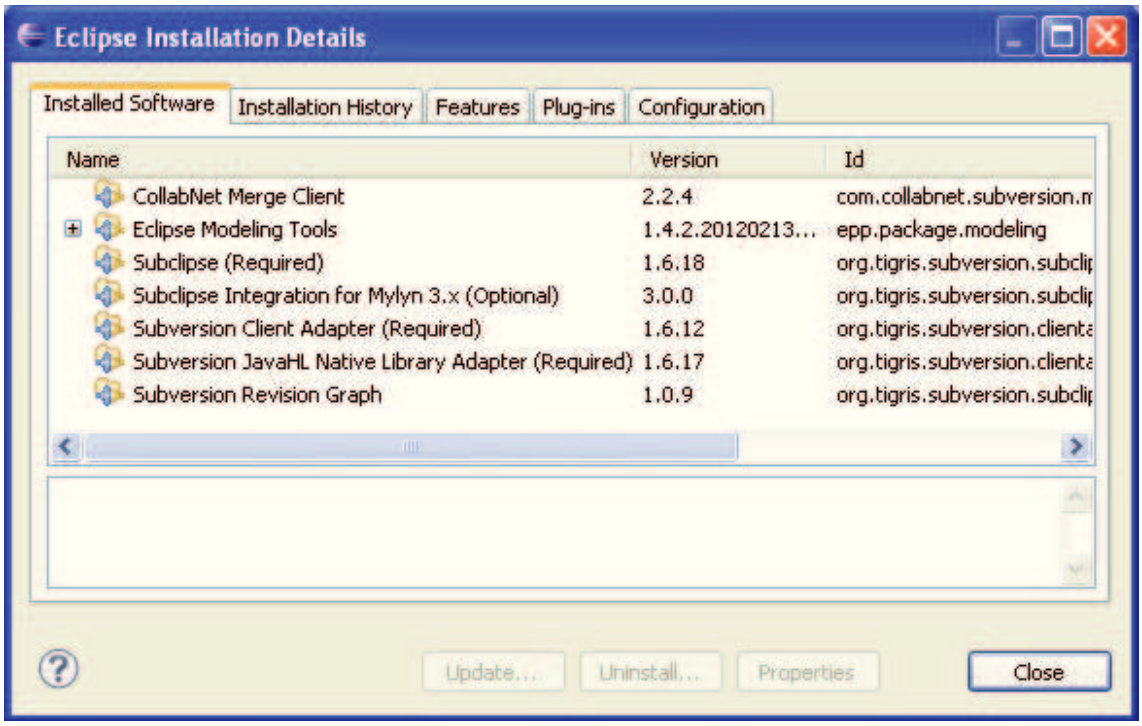


Obrázek 3: Workspace

Zvolil jsem vhodné umístění a potvrdil tlačítkem „OK“.

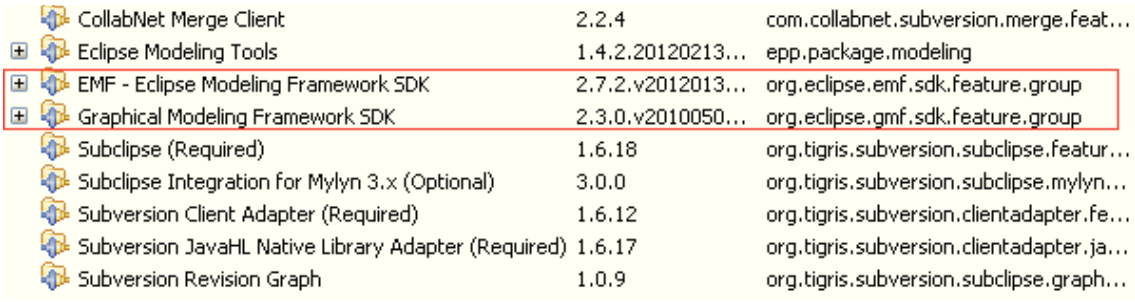
Klíčovým krokem bylo doinstalovat potřebné pluginy pro práci s GMF a EMF. Nejprve jsem zjistil, zda je již nemám nainstalované. V menu jsem vybral položku „Help“ a přešel na „Install New Software …“. Otevřelo se mi okno „Install“ (viz. Obrázek 32), v jehož pravé spodní části se nachází odkaz „already installed“.

Ten mě odkázal na okno „Eclipse Installation Details“, kde jsem viděl aktuálně nainstalovaný software.



Obrázek 4: Eclipse Installation Details

Jelikož nebyl nainstalovaný žádný software pro práci s GMF, zavřel jsem stávající okno a vrátil se zpět na okno instalace zobrazené na Obrázek 32. Zde jsem zvolil, že chci zobrazit všechny dostupné stránky. Toho jsem dosáhnul tak, že jsem vybral možnost „All Available Sites“ a tím jsem získal seznam veškerého software, který aktuálně byl k dispozici. Zvolil jsem tedy „EMF – Eclipse Modeling Framework SDK“ a „Graphical Modeling Framework SDK“ ze skupiny „Modeling“ a nainstaloval. Po průchodu průvodce a dokončení instalace mě Eclipse vybídnul k restartu. Jakmile se znovu spustil, mohl jsem se přesvědčit, že je vše nainstalováno jak má.

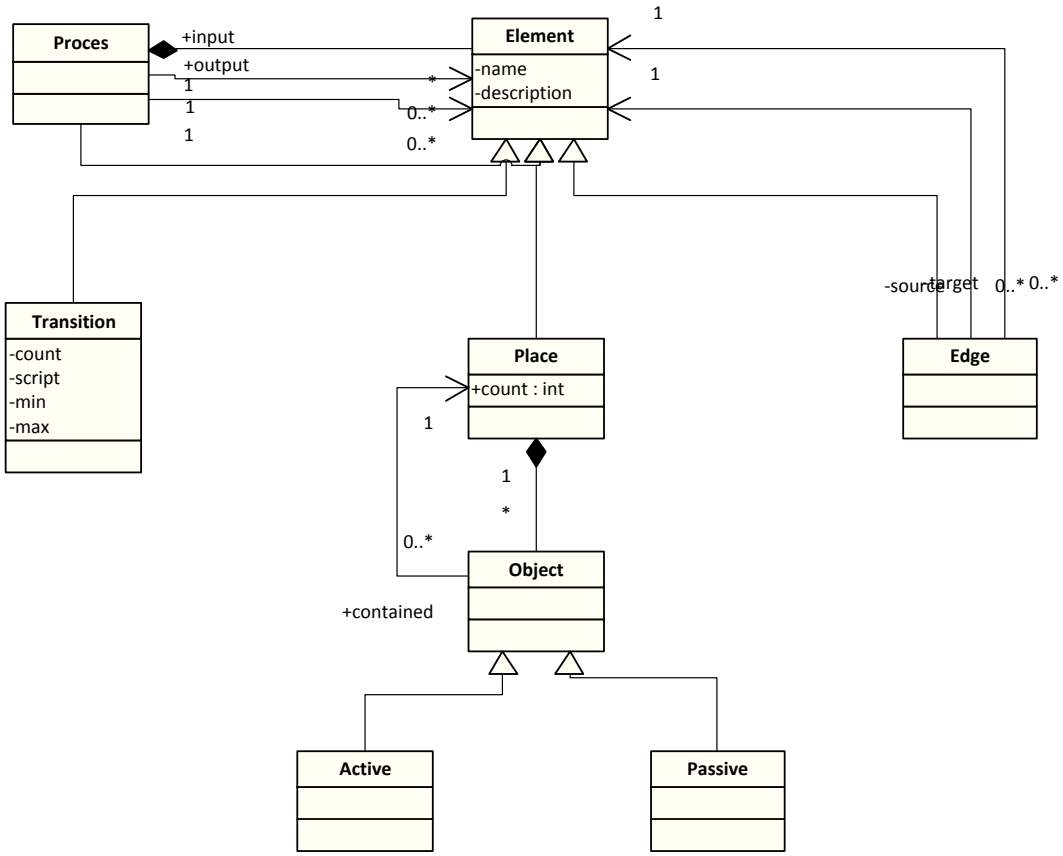


Obrázek 5: Nově nainstalovaný software

Nyní jsem měl Eclipse připraven pro práci s GMF i s podporou všech jeho funkcí, jako je např. grafický editor pro modelování metamodelu či nástroj GMF Dashboard, o kterých se zmíním v kapitole „Tvorba metamodelu“.

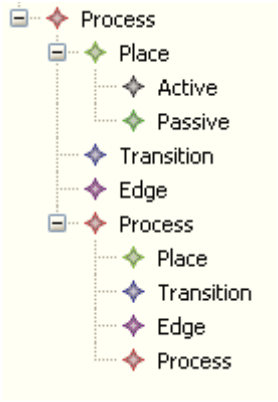
3.4.5 Analýza

Analýza je důležitou součástí každého projektu, proto i my (já a řešitel 2. části diplomové práce) jsme se rozhodli pro její realizaci. Není příliš rozsáhlá, ale nejdůležitější části pro realizaci projektu přeci jenom obsahuje. Jednou z nich je bezpochyby specifikace zadání. Toto jsem však již uvedl v kapitole „Cíle práce“, proto se budu zabývat spíše další částí analýzy a tou je diagram tříd.



Obrázek 6: Analýza – diagram

Na Obrázek 2 je uveden třídní diagram zrealizovaný v analýze. Základním elementem je třída Process, neboť právě Process na své nejvyšší úrovni tvoří samotné plátno diagramu. Pokud se odvolám na výše uvedenou definici agregace, tak Process znázorňuje celek a všechny elementy (včetně sebe sama) tvoří jeho části. Pokud Process není na nejvyšší úrovni, může být jak celkem, tak i něčí částí a nazývá se Subprocess. Podobně jako ve vztahu Process - Element se to má i ve vztahu Place – Object s tím rozdílem, že Place nemůže být částí sebe sama. Znázorním-li tuto situaci pomocí stromové struktury, bude to vypadat následovně:



Obrázek 7: Analýza – příklad stromové struktury modelu

Ještě vysvětlím význam jednotlivých asociací:

input	vstupní elementy podprocesu
output	výstupní elementu podprocesu
source	ukazatel na zdrojový element hrany
target	ukazatel na cílový element hrany
contained	ukazatel na místo, jehož součástí je daný objekt

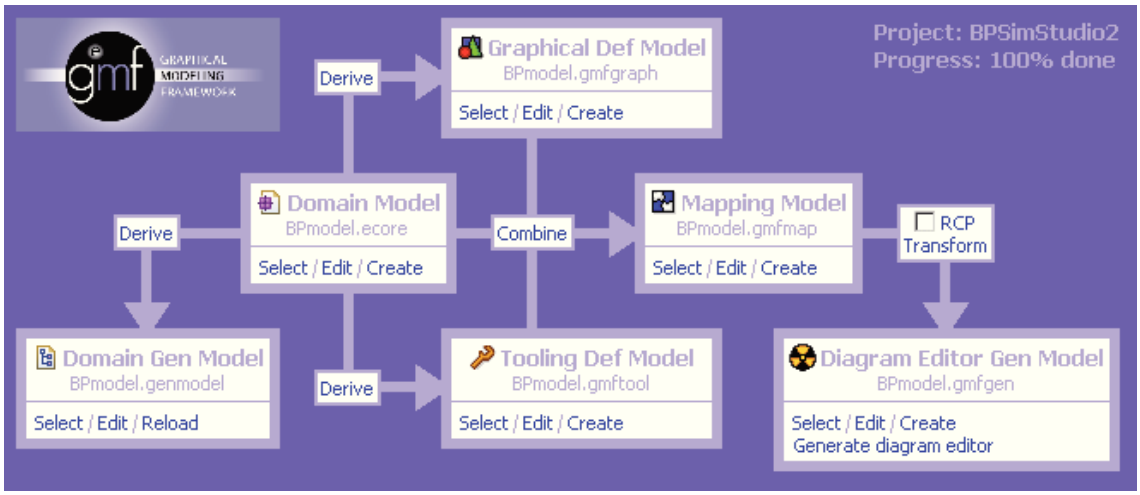
Tabulka 1: Význam asociací (1/2)

3.4.6 Tvorba metamodelu

Nyní jsem již měl jasno v tom, co vlastně chci tvořit, jak to mám tvořit a měl jsem k tomu i funkční vývojové prostředí. Mohl jsem tedy začít vyvíjet samotnou aplikaci. V první části to byla spíše práce s myší. Podstatou bylo vytváření skupiny XML souborů definujících celý metamodel, ale provádí se tak v přehledných grafických nástrojích, takže je práce mnohem snazší než při psaní kódu v textových editorech.

Začal jsem tedy tvořit projekt. V menu „File“ jsem vybral možnost „New Project“ a z nabídnutých možností zvolil „New GMF Project“. Jako název projektu jsem určil „BPSimStudio2“, který vychází z již existujícího editoru pro tvorbu a simulaci podnikových procesů. Vytvořený projekt obsahuje mimo standardní složky jako většina projektů ještě složku „model“ vyhrazenou pro soubory metamodelu.

Již výše jsem se zmínil o nástroji GMF Dashboard. Jedná se o pomocný nástroj při tvorbě GMF aplikací. Zobrazuje jednotlivé kroky procesu a vede jeho uživatele od počátečního vytvoření metamodelu k závěrečnému vygenerování zdrojového kódu editoru.



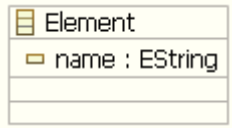
Obrázek 8: GMF Dashboard

V menu pod položkou „Window“ – „Show View“ – „Other…“ jsem zvolil GMF Dashboard. Metamodel označený prvkem „Domain Model“ mohu buď přiřadit z již existujících, nebo vytvořit nový. Vybral jsem tedy možnost „Create“ a po dokončení průvodce má složka „model“ obsahoval první soubor s názvem „BPmodel.ecore“. Otevřel jsem jej dvojklikem a v okně „Properties“ jsem nastavil následující vlastnosti:

Property	Value
Name	modelBP
Ns Prefix	modelBP
Ns URI	http://modelBP

Obrázek 9: BPmodel.ecore - Properties

Metamodelu jsem přiřadil jméno a doménu, ale stále chybělo vytvořit diagram definující výsledný model aplikace. Toho lze dosáhnou za pomoci příjemného grafického nástroje pro modelování diagramů,



Obrázek 10: Grafický náhled na metamodel

dále lze použít strukturální náhled na daný diagram,

Obrázek 11: Strukturální náhled na metamodel

anebo je možné model vytvořit psaním složitého XML souboru, který diagram reprezentuje.

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="modelBP"
    nsURI="http://modelBP" nsPrefix="modelBP">
  <eClassifiers xsi:type="ecore:EClass" name="Element">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
      eType="ecore:EDataType
      http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
</ecore:EPackage>
```

Výpis 1: Metamodel psaný v XML

Zvolil jsem si tedy 1. možnost. Kliknul jsem pravým tlačítkem myši na soubor metamodelu, vybral položku „Initialize ecore_diagram diagram file“. Po dokončení průvodce se mi zobrazil jednoduchý editor obsahující plátno a paletu elementů vyobrazený na Obrázek 33. Pro práci s tímto nástrojem je však potřeba znát alespoň základy tvorby UML diagramů, konkrétně objektového modelu. Vše potřebné pro pochopení této problematiky jsem již uvedl v kapitole „3.4.2 Základy UML“.

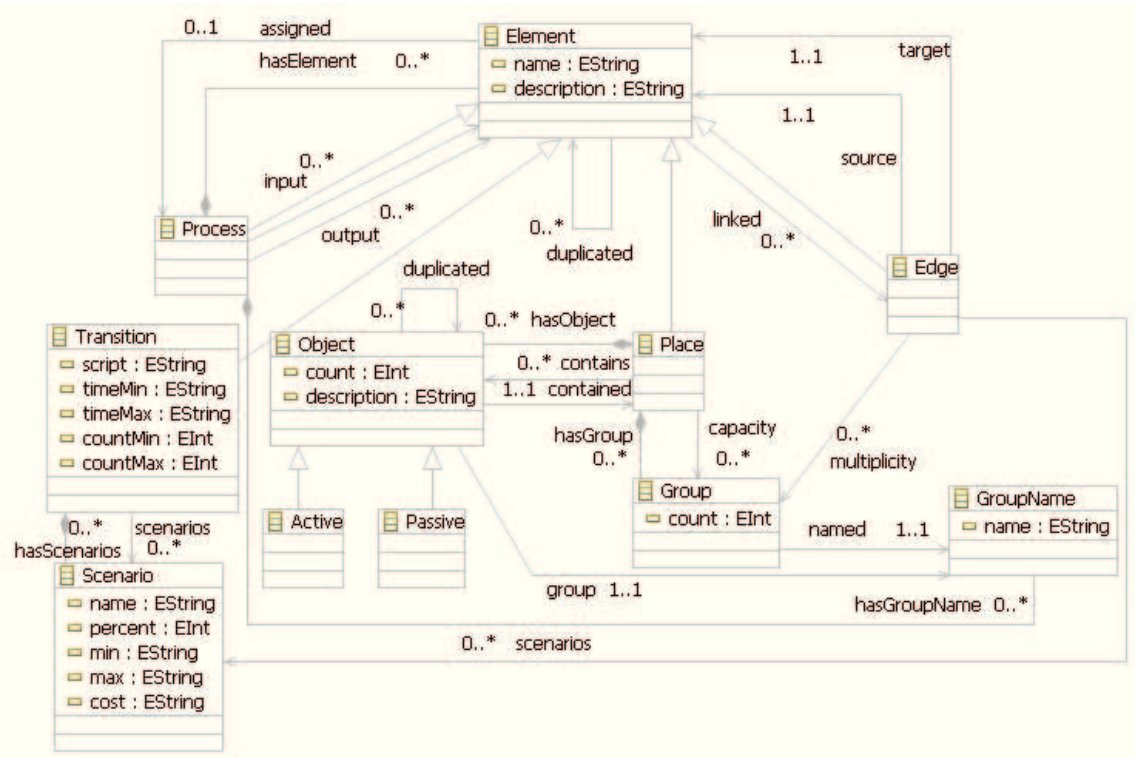
Ecore model umožňuje definovat tyto objektové prvky:

EPackage	prvek reprezentující root modelu
EClass	představuje třídu obsahující libovolný počet atributů a referencí.
EAttribute	představuje atribut, který má jméno a typ.
EDataType	představuje typ atributu jako např. int, float nebo Date
EOperation	operace patřící objektům, přidá funkcionalitu metamodelu
EAnnotation	poznámky objektů
EEnum	výčtový typ prvků
EEnumLiteral	prvek, který má jméno a hodnotu
Association	popisuje skupinu spojení mezi objekty
Aggregation	popisující vztah mezi celkem a jeho částmi
Generalization	vztah analogický dědičnosti

Tabulka 2: Výčet prvků Ecore modelu

O struktuře objektového modelu aplikace jsem měl již určitou představu po dokončení analýzy. Teď už jenom stačilo model zhotovit ve zmiňovaném editoru a provést drobné úpravy. Avšak při prvních pokusech o vygenerování aplikace jsem zjistil, že model má ještě mnohé nedostatky, a tak se tento model postupně dotvaroval do finální podoby až po několika pokusech. V kapitole „Analýza“ jsem již objasnil navrženou strukturu objektového modelu navrženého ve fázi analýzy, proto nyní plyně navážu na předchozí informace a objasním i výsledný objektový model. Záměrně píši „navážu“, neboť tyto 2 modely se mezi sebou liší pouze o několik rozšíření. Řečeno slovy matematika: „Graf A je podgrafem grafu V, jestliže graf A znázorňuje diagram analýzy, graf V znázorňuje výsledný diagram, třídy diagramů reprezentují vrcholy grafů a veškeré reference diagramů reprezentují hrany grafů.“

Na Obrázek 12 se můžete sami přesvědčit o mém tvrzení:



Obrázek 12: BPmodel.ecore_diagram

Podnětem pro 2 nejvýraznější rozšíření modelu bylo především přidání skupin a scénářů do aplikace. Bylo to způsobeno hlavně mou nedůsledností při specifikaci zadání projektu od zadavatele a následné analýze. Nedostatky byly doplněny až v pozdější fázi vývoje, což významně narušilo dosavadní strukturu aplikace. Změny v modelu sice nejsou příliš razantní, ale velkou nevýhodou GMF je, že sebemenší změna v metamodelu se ve výsledné aplikaci projeví až po opětovném průchodu celého procesu definic, transformací a generování zdrojových kódů editoru.

Přidání tabulky Scenario tedy zajistilo editoru možnost vytváření libovolného množství scénářů v přechodech a jejich přiřazení k připojeným hranám. V případě skupin je to trochu komplikovanější. Process na nejvyšší pozici obsahuje názvy skupin (GroupName), na které se odkazuje třída Group, která vždy definuje dvojici počet – jméno. Tímto způsobem přiřadím místům kapacitu u jednotlivých skupin (atribut „count“ reprezentuje maximální počet objektů dané skupiny v konkrétním místě) a hranám jejich násobnost v závislosti na skupinách (atribut „count“ reprezentuje maximální počet objektů dané skupiny, který hrana propustí). Na názvy skupin se také odkazují jednotlivé objekty, které tak definují, do které skupiny patří.

Přibyly také některé nové asociace mezi jednotlivými třídami, proto je opět shrnu do přehledné tabulky, jak tomu bylo v kapitole „Analýza“:

assigned	odkaz na rodičovský element
duplicated	seznam všech duplikátů daného elementu
linked	seznam hran vstupujících či vystupujících z elementu
contains	seznam objektů v místě
capacity	omezení místa na konkrétní skupiny a jejich kapacity
multiplicity	omezení násobnosti hran pro jednotlivé skupiny
named	přiřazení jména skupiny skupině
group	určení skupiny objektu
scenarios	přiřazení seznamu scénářů

Tabulka 3: Význam asociací (2/2)

Když jsem měl Ecore diagram konečně hotový, mohl jsem pokračovat dalším krokem v nástroji GMF Dashboard z Obrázek 8. Tím bylo vytvoření doménového modelu, jež se ukládá s koncovkou *.genmodel*. Kliknul jsem na „Derive“ směřujícího k bloku nazvaném „Domain Gen Model“. Po vytvoření souboru *BPmodel.genmodel* jsem jej otevřel, pravým tlačítkem myši kliknul na nejvyšší element zobrazené stromové struktury a zvolil „Generate All“. Tím se mi vygenerovaly:

- třídy jednotlivých objektů modelu ve složce *BPSimStudio2.src*
- projekt *BPSimStudio2.edit*
- projekt *BPSimStudio2.ediort*
- projekt *BPSimStudio2.tests*

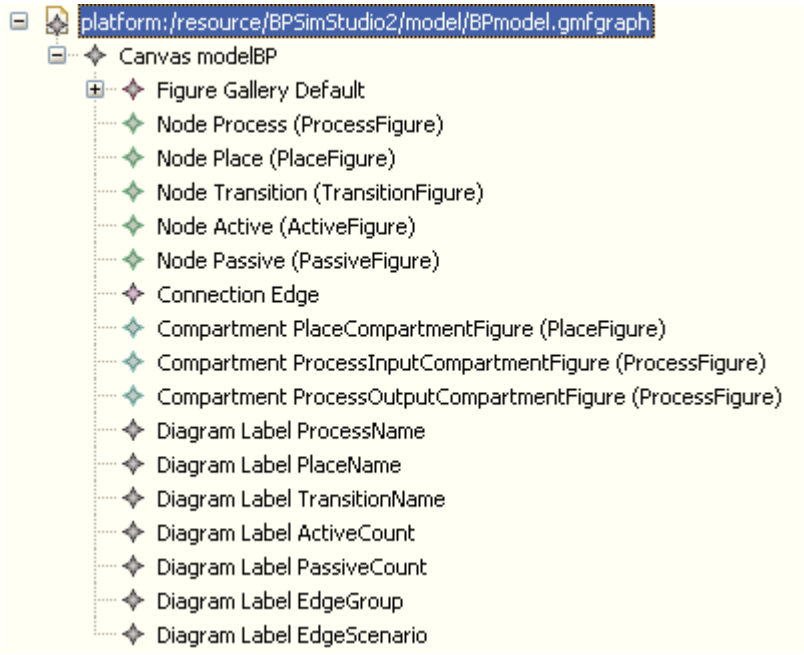
3.4.7 Grafická reprezentace editoru

V předchozí kapitole jsem popisoval, jak jsem získal model a vygeneroval jeho třídy, předmětem této kapitoly bude tvorba vzhledu editoru a jednotlivých elementů. Tento grafický model mé koncovku *.gmfgraph* a získal jsem jej odvozením od „Domain Model“, jak je patrné z GMF Dashboard na Obrázek 8. Toto samotné odvození však nebylo tak jednoduché, proto zmíním i to, jak jsem získal samotný soubor modelu. Po kliknutí „Derive“ mezi „Domain Model“ a „Graphical Def Model“ se zobrazila klasická výzva k zadání jména a umístění souboru, jako již několikrát při tvorbě všech předchozích typů modelů. V následujícím okně jsem měl vybrat element reprezentující plátno diagramu. Jak jsem již zmiňoval dříve, jedná se o element Process. Tlačítkem Next jsem se dostal na poslední okno průvodce, kde jsem konečně

mohl vybrat jednotlivé prvky modelu, které se mají zobrazovat a které zajišťují pouze funkcionalitu. Na výběr jsem měl z 3 skupin zobrazení:

Obrázek 13: Skupiny zobrazení (uzel – hrana – atribut)

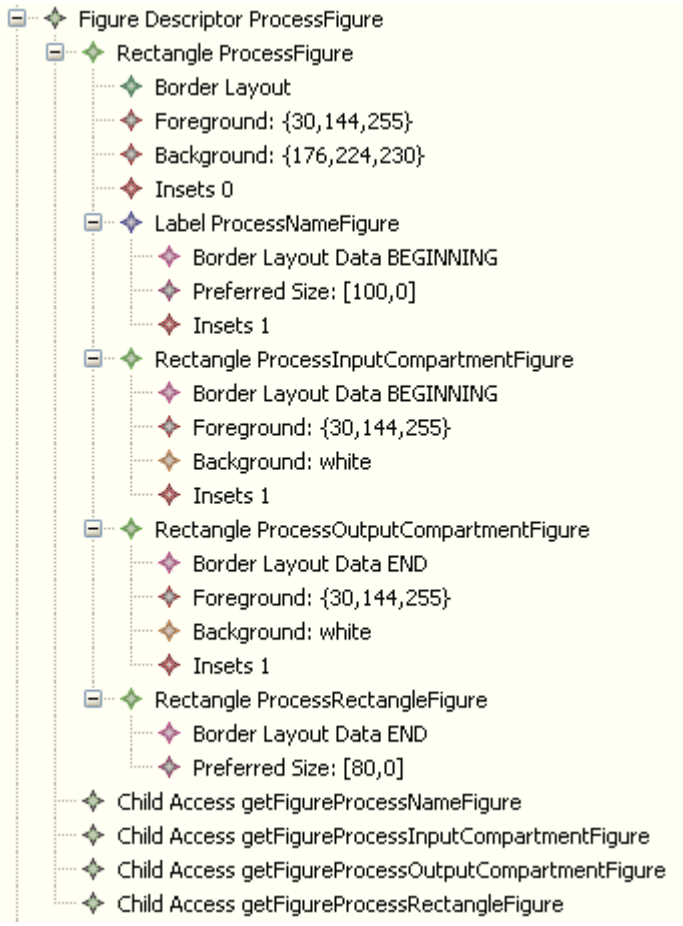
Které prvky jsem vybral je patrné z Obrázek 34 a Obrázek 35, které vyobrazují zatržení všech potřebných elementů. Po dokončení průvodce vypadal výsledný diagram následovně:



Obrázek 14: BPmodel.gmfgraph – Canvas

Z obrázku je patrné, že v souboru se mi promítly pouze ty elementy, které jsem si zvolil v průvodci. Prvky Node označují uzly, Connection hrany a Diagram Label jsou atributy, které se budou u zvolených elementů zobrazovat. Dále jsem musel přidat prvky Compartment, jejichž důvod jsem již dříve zmiňoval. Ještě jsem však musel určit, kde a jak se mají jednotlivé elementy diagramu zobrazovat a právě k tomu slouží poslední nezmíněné prvky označené jako Figure Descriptor obsažené v prvku Figure Gallery Default.

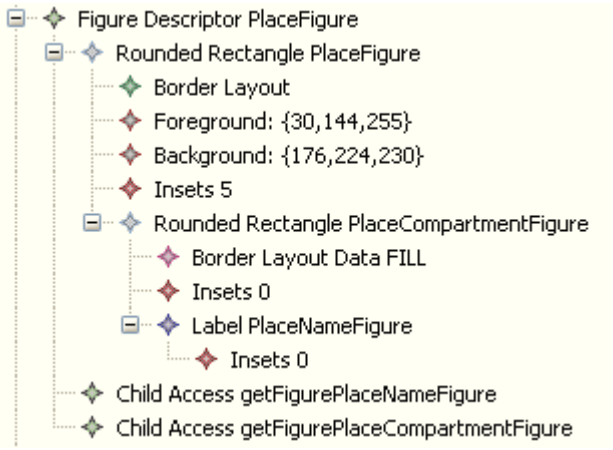
Figure Descriptor ProcessFigure:



Obrázek 15: Figure Descriptor ProcessFigure

Tento prvek definuje grafickou reprezentaci elementu Process. Základnu elementu tvoří obdélník (prvek Rectangle). Pro uspořádání jednotlivých prvků v elementu jsem zvolil BorderLayout, dále jsem nastavil tyrkysovou barvu pozadí a modrou barvu rámečku a šířku okrajů jsem nastavil na 0. Process obsahuje popisek se jménem (Label ProcessNameFigure), compartment pro vstupní a výstupní elementy (Rectangle ProcesInputCompartmentFigure, ProcesOutputCompartmentFigure) a obdélník ProcessRectangleFigure, který má pouze dekorační význam. Z jeho vlastností je patrné, že je umístěn ve spodní části elementu, má šířku 80 pixelů a výšku nulovou, tedy jeho hlavním úkolem je udržovat preferovanou šířku celého elementu. ProcessNameFigure vyplňuje horní část elementu a šířku okrajů jsem nastavil na 1 pixel. Posledními prvky elementu proces jsou vstupní a výstupní compartment. Ty jsem umístil hned pod Label se jménem na levou a pravou stranu. Barvu jejich rámečku jsem nastavil stejně, jako je barva rámečku celého elementu a pozadí bílé. Posledním nezmíněným typem prvku je Child Access. Ten slouží pouze pro přístup k jednotlivým prvkům.

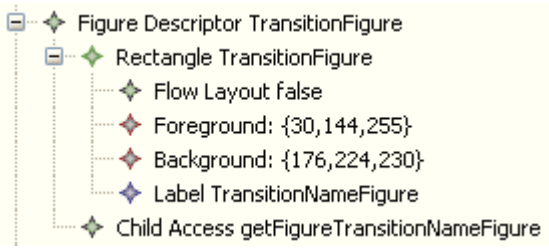
Figure Descriptor PlaceFigure:



Obrázek 16: Figure Descriptor PlaceFigure

Základnu tohoto elementu tvoří obdélník se zaoblenými rohy. Je sice zvykem značit místa v PN kruhem, ale od toho jsem však musel upustit, protože vkládat prvky do kruhového elementu je problém. Border Layou pracuje s obdélníky, ale nebere v potaz to, že se má nějaký element vykreslovat jako kruh. Když např. text začínal v levém horním rohu, tak se ve výsledku vykreslil mimo kruhovou oblast elementu. Po domluvě s vedoucím práce jsem tedy usoudil, že nejpřijatelnějším tvarem pro Place bude zaoblený obdélník. Při větším zaoblení budí dojem kruhu a zároveň s ním dobře pracuje Border Layout. Barvu pozadí a rámečku již nebudu zmiňovat, jelikož je stejná jako u většiny elementů. Zmíním však šířku okrajů, kterou jsem nastavil na hodnotu 5, díky čemuž mi text nepřesahuje přes zaoblené rohy. Základní zaoblený obdélník jsem vyplnil dalším zaobleným obdélníkem, jenž tvořil compartment pro vkládání objektů do místa. Label s názvem místa jsem v tomto případě musel vložit přímo do compartment, opět z estetického důvodu.

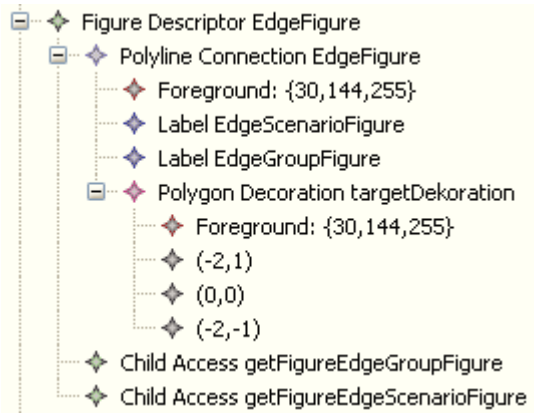
Figure Descriptor TransitionFigure:



Obrázek 17: Figure Descriptor TransitionFigure

Transition je vyobrazen jako tyrkysový obdélník s modrými hranami a ve své horní části má umístěn název.

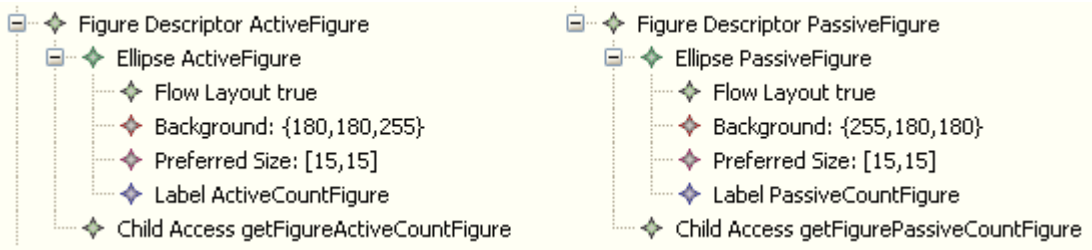
Figure Descriptor EdgeFigure:



Obrázek 18: Figure Descriptor EdgeFigure

Tento element tvoří hrany mezi místy, přechody a podprocesy. Je to tedy spojnice, jejíž jedna část je zakončena šipkou. Edge (hrana) je modré barvy a osahuje 2 nápisy (Label). Jeden vyobrazuje scénáře a druhý skupiny. Avšak tuto funkcionalitu jsem již nestihl dokončit, proto se u každé hrany zobrazuje pouze malá ikona. Prvek targetDecoration vykresluje šipku na konci hrany. Je definován třemi body, jež tvoří oblast ohraničenou rovnoramenným trojúhelníkem. Hlavní vrchol trojúhelníku je v bodě (0,0), což jsou souřadnice místa označujícího konec hrany. Celá oblast se při zobrazení vyplní modrou barvou, což ve výsledku dodá vzhled šipky.

Figure Descriptor ActiveFigure, PassiveFigure:

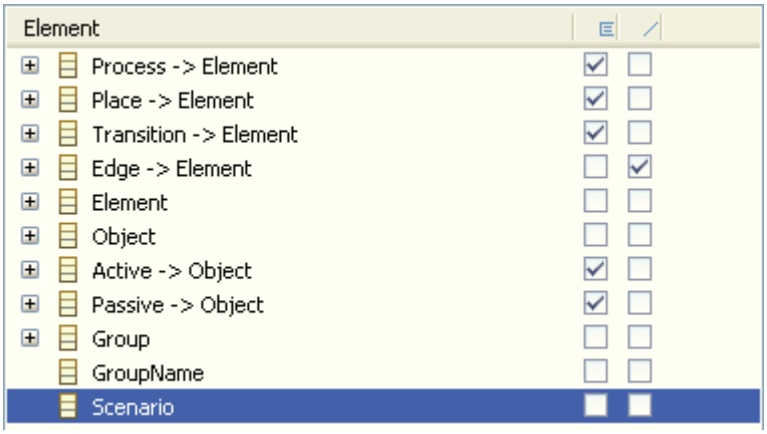


Obrázek 19: Figure Descriptor ActiveFigure, PassiveFigure

Nakonec jsem upravil aktivní a pasivní objekty. Jsou tvořeny elipsou s preferovanou šířkou a výškou 15 pixelů a textem vyobrazujícím hodnotu atributu count (počet výskytů daného objektu). Barva aktivního objektu je odlišná od barvy pasivního.

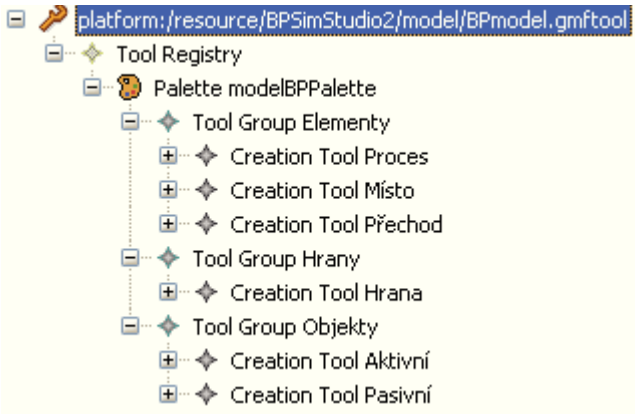
3.4.8 Paleta nástrojů

Když jsem dokončil grafickou specifikaci editoru, pustil jsem se do specifikace palety nástrojů. Kliknutím na „Derive“ mezi „Domain Model“ a „Tooling Def Model“ jsem spustil průvodce, při dotazu na element reprezentující plátno diagramu jsem opět vybral Process, až jsem se dostal ke specifikaci prvků, které chci zobrazit v paletě nástrojů. Obrázek 20 zobrazuje můj výběr.



Obrázek 20: Tooling Definition

Dokončil jsem průvodce a získal tak soubor *BPmodel.gmftool*. Provedl jsem několik úprav a výsledkem byl tento obsah souboru:

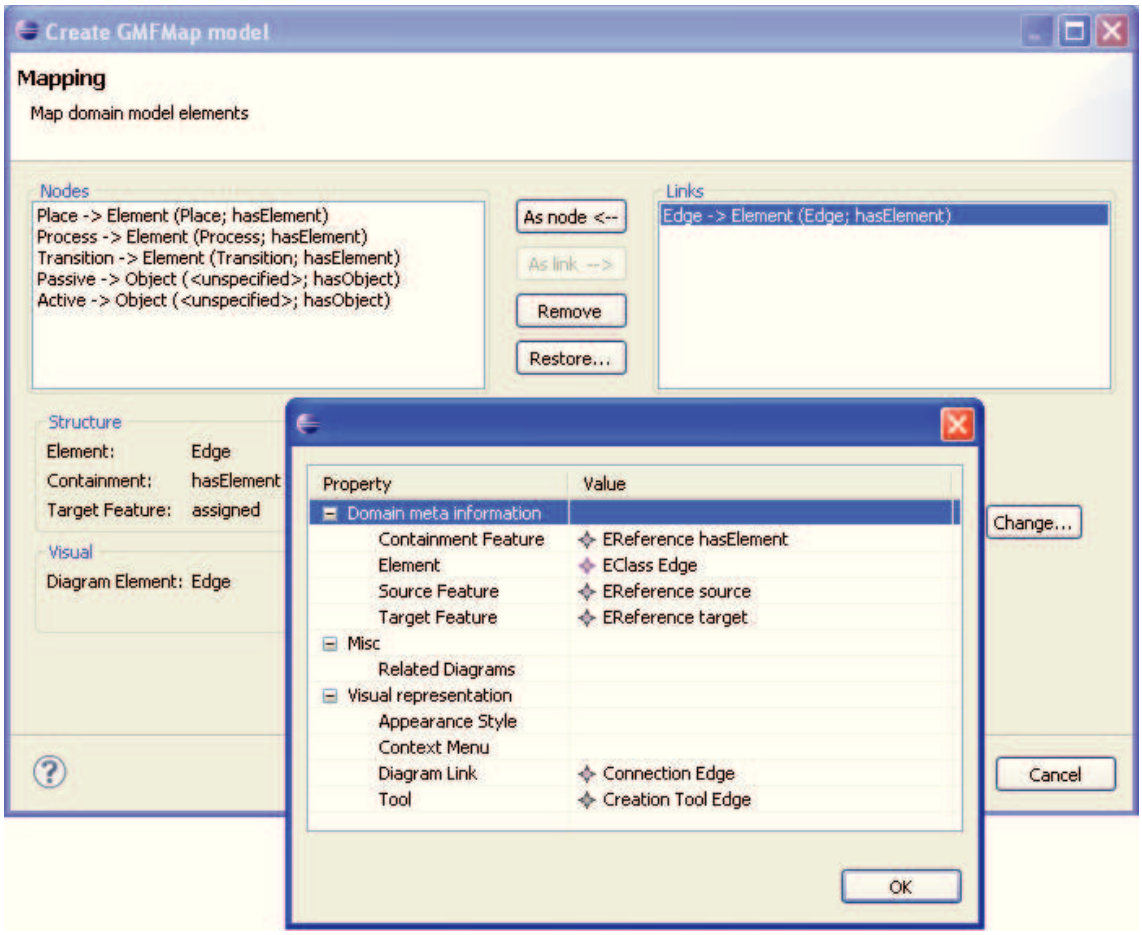


Obrázek 21: BPmodel.gmftool

Prvek Palette představuje paletu nástrojů. V té jsem vytvořil 3 skupiny nástrojů, a to Elementy, Hrany a Objekty. Skupina Elementy obsahuje prvky, které tvoří jednotlivé uzly PN, tedy podproces, místo a přechod. Skupina Hrany obsahuje prvky tvořící hrany PN a nakonec skupina Objekty obsahuje objekty PN, tedy aktivní a pasivní objekty.

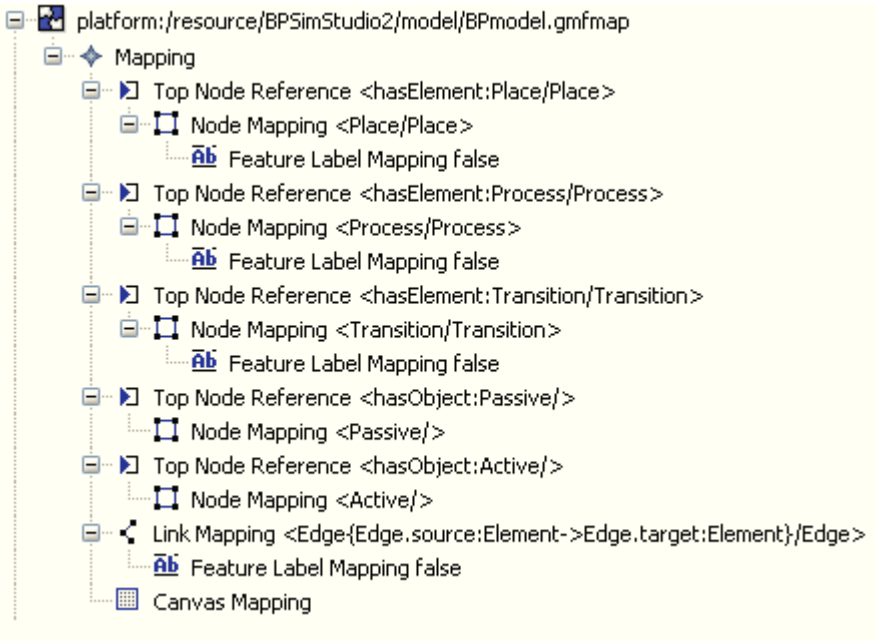
3.4.9 Definice vztahů mezi objektovou a grafickou reprezentací

Jakmile jsem dokončil grafickou definici a definici nástrojů, GMF Dashboard mi nabídl možnost učinit další krok, a to tvorbu modelu mapování. Kliknutím na „Combine“ jsem vyvolal průvodce, který sjednotí předchozí 3 modely a vytvoří model mapování. Opět jsem vybral Process jako prvek reprezentující diagram, v dalších krocích jsem určil zdrojové modely pro vznik výsledného a nakonec jsem se dostal k samotnému mapování.



Obrázek 22: Mapping

V tomto kroku definuji, jak se jednotlivé elementy budou chovat, kterým nástrojem se vytváří nebo který grafický prvek reprezentuje který element modelu. Mezi uzle (Nodes) jsem zařadil Place, Process, Transition, Passive a Active. Poslední element Edge jsem označil jako spojnici (Links). Ten však neměl defaultně nastavené potřebné vlastnosti, proto jsem je musel změnit. Tlačítkem „Change…“ jsem otevřel okno vlastností a nastavil podle Obrázek 22. Upozorním hlavně na vlastnost Tool, která může být defaultně přednastavená např. na „Creation Tool Place“. V případě, že bych zapomněl hodnotu změnit, tak se nic neděje, protože po vygenerování modelu jsem musel překontrolovat tuto vlastnost i u všech ostatních elementů.



Obrázek 23: BPmodel.gmfmap – bez úprav

Prošel jsem tedy všechny prvky modelu a v části vlastního mapování (Node Mapping) zkontroloval, případně opravil, hodnotu atributu Tool. To byla ta jednodušší část, ale pak jsem musel rozvrhnout hierarchii jednotlivých prvků tak, aby seděla s danou filozofií editoru (viz. Obrázek 36).

Přímo na plátno diagramu mohu vkládat pouze elementy Place, Process, Transition a Edge, proto jsou obsaženy přímo v prvku „Mapping“. Jenomže Place a Transition mohu vytvářet také v compartment Process, a to jak vstupním tak výstupním, proto jsem musel vytvořit reference (Child Reference) na již existující uzle Place a Transition a ty přiřadit k jednotlivým

compartment (Compartment Mapping) elementu Process. Např. pro místo přiřazené výstupnímu compartment má následující vlastnosti:

Property	Value
Compartment	Compartment Mapping <ProcessOutputCompartmentFigure>
Containment Feature	Process.hasElement:Element
Child	Node Mapping <Place/Place>
Children Feature	Process.output:Element
Referenced Child	Node Mapping <Place/Place>

Obrázek 24: Properties - BPmodel.gmfmap - output: Place

Posledními nezmíněnými elementy jsou Active a Passive. Tyto se nevkládají přímo na plátno editoru, ale pouze do elementu Place, proto jsem jejich reprezentaci musel přesunout z prvku „Mapping“ do prvku reprezentujícího Place. Opět jsem je přiřadil k jednomu compartment s názvem PlaceCompartmentFigure reprezentujícího oblast pro vkládání objektů do místa.

Ještě zmíním prvek „Feature Compartment Mapping“, který jednotlivým textovým polím elementů editoru přiřazuje atribut, který se bude zobrazovat, případně editovat.

Vytváření modelu mapování je nejvýznamnějším krokem celého mapování, protože právě zde se skládají dříve vytvořené části metamodelu do jednoho celku tvořícího logiku celého editoru.

3.4.10 Vygenerování aplikace

Dostal jsem se k poslednímu kroku před prvním otestováním funkční aplikace. Pomocí GMF Dashboard jsem provedl transformaci nad modelem mapování a získal tak model pro generování editoru. Jak již název napovídá, je to výchozí model pro vygenerování zdrojového kódu digramu.

Jelikož je to pouze souhrn jednotlivých částí metamodelu (viz. Obrázek 37), tak nebudu podrobně probírat každou jeho část, pouze uvedu poslední změny, které jsem musel udělat.

Zatím v celém projektu jsem nikde nedefinoval, jak vlastně bude fungovat vytváření podprocesů. Musel jsem určit, že tuto funkcionalitu zajišťuje Process a že se podproces bude chovat jako nové plátno diagramu. Podproces reprezentuje prvek „Gen Top Level Node Process2EditPart“. Číslo „2“ muselo být přidáno, jelikož prvek „ProcessEditPart“ již reprezentuje samotné plátno editoru na nejvyšší úrovni. Přidáním prvku „Open Diagram Behaviour“ určím, že právě tento element vytváří podprocesy a dvojklikem na něj se má otevřít nové okno reprezentující plátno podprocesu. To, jak se bude nové okno editoru chovat, určují vlastnosti „Diagram Kind“ a „Editor ID“. Pokud není nastavena žádná hodnota, jak tomu je v tomto případě, tak se nové okno podprocesu bude chovat stejně jako plátno na nejvyšší úrovni. Pokud však chceme nastavit, že plátnem je jiný element metamodelu, nebo nějaký

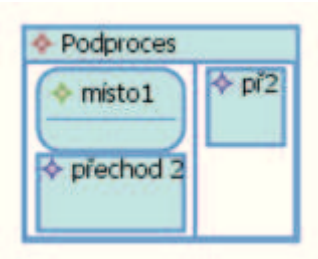
element z jiného metamodelu, stačí přiřadit výše uvedeným vlastnostem identifikační údaje daného elementu. Pokud bych např. nastavil element Place jako plátno podprocesu, pak bych v novém okně mohl pouze vytvářet aktivní a pasivní objekty místo stávajících míst, přechodů, podprocesů a hran. Touto problematikou se detailněji zabývá [4] Jevon Wright v kapitole „GMF Diagram Partitioning“.

Druhou a zároveň poslední úpravou tohoto modelu bylo uspořádání objektů v compartment místa. Stačilo pouze vlastnost „List Layout“ prvku „Gen Compartment PlacePlaceCompartmentFigureEditPart“ změnit na hodnotu „true“ a tím jsem docílil, že se všechny objekty v místě budou řadit úhledně pod sebe.

Nakonec jsem model uložil a pomocí GMF Dashboard vygeneroval zbývající třídy potřebné pro spuštění editoru. V tomto okamžiku již bylo možné aplikaci spustit a vyzkoušet si základní funkcionalitu editoru. Jak spustit výslednou aplikaci uvedu v kapitole „Testování“.

3.4.11 Uspořádání elementů v compartment procesu

Jednou z nejdůležitějších úprav, kterými jsem se musel zabývat, bylo uspořádání elementů v compartment procesu. Kdybych se touto problematikou nezabýval, výsledkem by mohl být např. takovýto výstup:



Obrázek 25: Element Process - bez úprav

To však není přípustné hlavně při větším množství elementů uvnitř podprocesu, a to jak z estetické stránky, tak i z hlediska použitelnosti. Proto jsem přišel s řešením, které šetří místo, dobře vypadá a použití je také jednoduché. Process bude mít nastavenou pevnou šířku a výšku, dokud nebude obsahovat žádný element. Při vložení každého dalšího elementu se Process roztáhne o výšku vloženého elementu a velikost jeho okraje. Vloženému elementu se upraví velikost na předdefinovanou výšku a šířku. V případě, že Process bude obsahovat velké množství elementů, bude možné jeho velikost změnit a mezi prvky bude možné listovat pomocí posuvníku.

Výsledek by měl vypadat následovně:

Obrázek 26: Element Process – upravený

Od návrhu jsem přistoupil k realizaci. Musel jsem vytvořit posluchače, který reaguje na změnu velikosti compartment. Ve chvíli, kdy je přidán nebo odebrán element z compartment, se změní velikost a zavolá se metoda, která upraví velikost a rozmístění jednotlivých elementů.

Místo v kódu, které nejlépe vyhovuje pro přidání posluchače na událost, je vygenerovaná metoda „createFigure()“ ve třídě „ProcessProcessInputCompartmentFigure“ pro vstupní a „ProcessProcessOutputCompartmentFigure“ pro výstupní compartment, která vytváří grafickou reprezentaci oblasti compartment.

```
/**
 * @generated NOT
 */
public IFigure createFigure() {
    ResizableCompartmentFigure result =
          (ResizableCompartmentFigure) super.createFigure();
    result.setTitleVisibility(false);
    // nastavení XYLayout
    IFigure contentPane = result.getContentPane();
    contentPane.setLayoutManager(new XYLayout());
    // přidání posluchače na událost zvětšení
    result.addFigureListener(new CompartmentFigureListener(this));
    return result;
}
```

Výpis 2: ProcessProcessIn/OutputCompartmentFigure.createFigure()

Celé metodě předchází komentář „@generated NOT“, aby generátor kódu věděl, že jí při opětovném generování má ponechat v původní podobě. Abych mohl s elementy v compartment jednoduše pracovat, musel jsem mu nastavit XYLayout, který umožňuje prvky vkládat pomocí zadaných souřadnic. Dále jsem přidal již zmiňovaného posluchače „CompartmentFigureListener“, který ve chvíli, kdy dojde ke změně velikosti compartment, zareaguje a provede patřičné úpravy. Ve třídě posluchače totiž přepisuji metodu „figureMoved(IFigure f)“, která je volána právě ve chvíli změny velikosti. Celý zdrojový kód

třídy označený jako Výpis 7 je mezi přílohami, proto zde v textu uvedu pouze část podstatnou pro vysvětlení problematiky.

```
List<AbstractEditPart> childs =
compartmentEditPart.getChildren();
int count = 0;
int resolution = elementSize + 2;
for (AbstractEditPart child : childs) {
  if (child instanceof AbstractGraphicalEditPart) {
    //element
    AbstractGraphicalEditPart gEditPart =
         (AbstractGraphicalEditPart) child;
    int x = is.left;
    int y = count * resolution + is.top;
    Rectangle constraint =
         new Rectangle(x, y, elementSize, elementSize);
    //přiřazení vzhledu elementu
    contentPane.setConstraint(gEditPart.getFigure(),constraint);
    count++;
  }
}
```

Výpis 3: CompartmentFigureListener - část metody figureMoved

Je to výňatek z metody „figureMoved(IFigure f)“, jež v argumentu předává objekt reprezentující compartment. Z něj získám samotný obdélník pojmenovaný „contentPane“, jemuž následně přiřadím patřičné elementy. Dále v kódu používám objekt „is“, který je typu „Insets“ a představuje velikosti okraje obdélníku. Posledním nedefinovaným prvkem kódu je „compartmentEditPart“, který je typu „ProcessProcessInputCompartmentFigure“. Ten jsem předal třídě jako argument konstruktoru při vytváření posluchače. Objekt „compartmentEditPart“ mi nyní poskytne jednotlivé elementy obsažené v daném compartment, které postupně procházím. Každému z nich nastavím velikost a pozici a nakonec přidám do „contentPane“. Pozici zadávám pomocí souřadnic „x“ a „y“. První souřadnici přiřadím hodnotu levého okraje, té druhé určím hodnotu podle vztahu: count * resolution + 1, kde count je pořadí vykreslovaného elementu počítající se od 0, resolution jsou rozestupy mezi jednotlivými elementy a vrchní okraj na konci přičítám kvůli odsazení. Velikost elementu mám pevně definovanou a je to 18 pixelů. Cyklus opakuji pro každý element a v každém průchodu zvyšuji hodnotu „count“ o 1.

Výsledek potom vypadá např. tak, jako na Obrázek 26.

3.4.12 Dialogová okna

Tato podkapitola pojednává o vytváření dialogových oken pro nastavení vlastností elementů diagramu, jak předávat potřebné argumenty a jak opět vkládat nastavené vlastnosti zpět do objektového modelu diagramu. Pro nastavení vlastností existuje v Eclipse okno vlastností (Properties), ale není zdaleka tak přehledné, jak by bylo vhodné a také poskytuje pouze omezenou funkcionalitu. Například vytváření nových skupin a scénářů není možné, proto jsem se rozhodl pro vlastní řešení. Ke každému elementu jsem vytvořil vlastní dialogové okno, které poskytuje uživateli přehled veškerých vlastností potřebných pro správné nastavení elementu a tím i celé sítě. Lze také nastavovat vlastnosti potřebné pro případnou simulaci. Dialogové okno uživatel zobrazí dvojklikem na příslušný element a po dokončení práce má možnost údaje odeslat k uložení nebo změny ignorovat.

Veškeré dění začíná v balíčku „modelBP.diagram.edit.parts“, kde dochází k úpravám všech částí vytvářeného diagramu. Pokud uživatel provede nějakou úpravu, vytvoří se požadavek (request), který prochází modelem a sbírá potřebná data. Nakonec se provedou všechny úpravy modelu, které request obsahuje a tím proces končí. V praxi to potom vypadá následovně. Při vytváření požadovaného prvku se volá metoda „createDefaultEditPolicies()“, kde se nastavují jednotlivé politiky, které prvku řeknou, které instance třídy EditPolicy mají být volány při nějaké akci.

```
protected void createDefaultEditPolicies() {
   // ...
   installEditPolicy(EditPolicyRoles.OPEN_ROLE,
        createOpenEditPolicy());
}

private OpenEditPolicy createOpenEditPolicy() {
   OpenEditPolicy policy = new OpenEditPolicy() {
     protected Command getOpenCommand(Request request) {
        // ...
        return new ICommandProxy(new UpdatePropertyCommand(
             (FontStyleImpl) link));
     }
   };
return policy;
```

Výpis 4: Přidání nové politiky prvku modelu

V mém případě prvku přiřazuji konkrétní instanci třídy OpenEditPolicy, která bude volána při dvojkliku na prvek. Politika OPEN_ROLE je obvykle instalována v případech, kdy může dojít ke změně prvku tím, že je otevřeno a modifikováno další okno. Jedná se zpravidla o otevření okna podprocesu dvojklikem, což je právě to, co potřebuji.

Když jsem před chvílí uváděl, že jsou volány instance třídy OpenEditPolicy, nebyl jsem tak úplně přesný. Tato třída totiž obsahuje metodu „getOpenCommand(Request request)“, která je volána v případě vyvolání příslušné akce (v tomto případě dvojkliku). Metoda v argumentu předává již výše zmiňovanou třídou Request. Z té získá informace o prvku, na kterém byla vyvolána akce a vrátí příkaz, že chce provést změny. Request jej zaznamená a ve fázi modifikace prvku zohlední i jej. Příkazem v přesném slova smyslu je myšlena třída „UpdatePropertyCommand“, která dědí od „AbstractTransactialCommand“ a obsahuje metodu „doExecuteWithResult“, která je volána právě ve fázi modifikace prvku.

```
protected CommandResult doExecuteWithResult(IProgressMonitor
     monitor,  IAdaptable info) throws ExecutionException {
   try {
      PlaceImpl element = getElement();
      PlaceAttributes place = new PlaceAttributes(element);
      EList<GroupName> groups = getGroups(element);
      // Vytvoření a otevření dialogového okna
      JFrame mainFrame = new JFrame("Title");
      mainFrame.setResizable(false);
      PlacetDialog dialog = new PlacetDialog(mainFrame, true, place,
           groups);

      if (dialog.getResult() == "OK") {
        place = dialog.getPlace();       // získání nových hodnot
        place.setPlaceImpl(element);     // promítnutí změn v elementu
      }
      return CommandResult.newOKCommandResult();
   } catch (Exception ex) {
      throw new ExecutionException("Can't open diagram", ex);
   }
}
```

Výpis 5: Vyvolání dialogového okna

Tato metoda „doExecuteWithResult“ demonstruje modifikaci místa (Place). Privátní metoda „getElement()“ vrací objekt třídy „PlaceImpl“, který reprezentuje objektový model místa. Ten je transformován do třídy „PlaceAttributes“, která představuje modifikovatelné atributy místa. Další privátní metodou je „getGroups(EObject element)“, která vrací seznam všech vytvořených skupin, které je možné přiřadit místu. Další část kódu spustí dialogové okno pro nastavení vlastností místa. Jak přesně vypadá třída „PlaceDialog“ reprezentující dialogové okno, uvedu později v této kapitole, podstatné však je, že po stisknutí tlačítka „OK“ vrátí nové hodnoty místa. Následně je model místa „PlaceImpl“ modifikován s novými hodnotami a metoda vrátí zprávu o úspěšném výsledku modifikace.

V textu jsem zmínil třídu „PlaceAttributes“. Tuto třídu jsem vytvořil kvůli jednoduššímu zpracování dat v dialogových oknech.

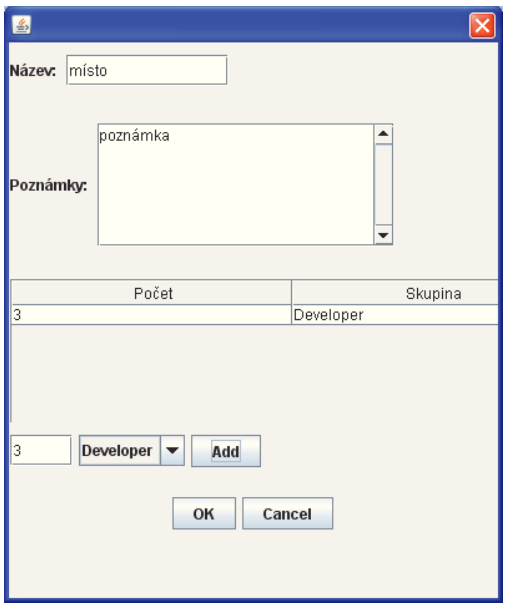
Obsahuje pouze atributy, které potřebuje uživatel u místa modifikovat, tedy jméno, poznámky a kapacitu. Dále obsahuje metodu „setPlaceImpl(PlaceImpl place)“, která promítne provedené změny zpět do třídy „PlaceImpl“.

```
public class PlaceAttributes {
   ElementAttributes element;
   EList<Group> capacity;

   public PlaceAttributes(PlaceImpl place) {...}

   public PlaceAttributes(ElementAttributes _element,
        EList<Group> _capacity) {...}
   public void setPlaceImpl(PlaceImpl place) {...}
}
```

Výpis 6: PlaceAttributes

„PlaceAttributes“ není jedinou třídou reprezentující atributy, neboť téměř každý element má svou vlastní. Jak je z kódu patrné, existuje i „ElementAttributes“, která obsahuje atributy name a description, které jsou společné pro všechny elementy dědící od třídy „Element“, proto místo těchto atributů uchovávám přímo třídu „ElementAttributes“.

V závěru kapitoly se dostáváme ke třídě „PlaceDialog“, která reprezentuje dialogové okno místa. Jedná se převážně o zápis GUI a množství pomocných metod a tříd, proto bude rozumnější místo zdrojového kódu uvést přímo obrázek.

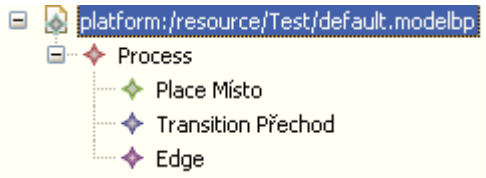


Obrázek 27: Dialogové okno vlastností místa

Okno obsahuje textové pole reprezentující atribut name (název), textovou oblest pro description (poznámky) a tabulku s hodnotami kapacit (capacity). Kapacita je reprezentovaná třídou „Group“, která obsahuje atribut count (počet) a odkaz na třídu „GroupName“, která obsahuje jméno skupiny. Atribut count uživatel zadá ručně, ale „GroupName“ musí vybrat z rozbalovacího seznamu, který je při inicializaci okna naplněn hodnotami ze seznamu skupin předaných třídě v argumentu konstruktoru. Tlačítko „Add“ vytvoří novou instanci třídy „Group“ se zadanými hodnotami a přidá do tabulky. Po stisknutí tlačítka „OK“ nebo „Cancel“ se okno ukončí a pomocí metody „getResult()“ předá informaci o ukončení. K získání modifikovaných dat z „PlaceDialog“ slouží metoda „getPlace()“.

3.4.13 Testování

O programování jednotlivých částí kódu bych se mohl rozepsat ještě na mnoho stránek, místo toho však ukážu, jak výslednou aplikaci spustit a otestovat, aby si čtenář případně mohl upravit editor dle vlastní fantazie sám. Jak jsem již zmínil v úvodu práce, výsledná aplikace je tvořena několika pluginy, které ke svému běhu potřebují prostředí Eclipse. Nejjednodušším způsoben, jak je v Eclipse spustit, je kliknout na jeden z pluginů pravým tlačítkem myši, přejít na položku „Run as“ a zvolit „EclipseApplication“. Spustí se nová instance Eclipse. Zde vytvořím nový projekt a do něj přidám prázdný diagram. Ten najdu opět v menu pod položkou „New“ a „Example…“. Vyberu BPmodel Diagram a dokončím průvodce. Nyní mám v projektu 2 nové soubory s koncovkami *.modelbp* a *.modelbp_diagram*. První ukládá objektový model vytvořeného diagramu a zobrazuje jej ve stromové struktuře…



Obrázek 28: Default.modelbp

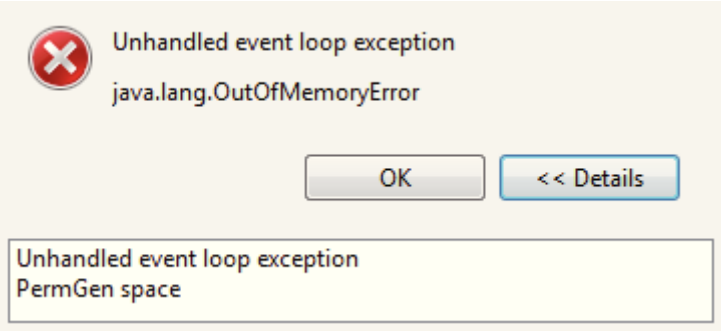
… a druhý reprezentuje diagram vytvořené sítě.

Obrázek 29: Default.modelbp_diagram

Nyní už mohu vytvářet libovolné diagramy, editovat skupiny nebo měnit vlastnosti jednotlivých elementů sítě.

3.5 Problémy

Počáteční problémy s nastavením prostředí jsem již ujasnil v kapitole „Příprava vývojového prostředí“, ale rád bych zde zmínil komplikaci, na kterou bychom mohli narazit i v dobře nastaveném prostředí. Jedná se o chybové hlášení o nedostatku paměti.



Obrázek 30: Chyba PermGen

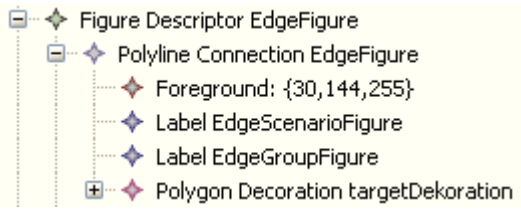
PermGen je speciální část paměti (haldy), která ukládá třídy, metadata, atd. Ta je defaultně nastavená na 64Mb. Překročíme-li její maximum, objeví se chybová hláška z Obrázek 30. Řešením je nastavit vyšší hodnotu maxima, což provedeme tak, že v menu v části „Run“ vypereme položku „RunConfigurations…“. Zde v části „EclipseApplication“ na záložce „Arguments“ vložím do pole označeného „VM arguments“ tento text: „-XX: MaxPermSize = 128M“. Tím dám příkaz virtuálnímu stroji javy, aby zvýšil maximální hodnotu PermGen paměti na 128Mb.

3.6 Možnosti rozšíření

Aktuální stav aplikace je takový, že je dostatečně funkční a připravená k použití. Splňuje většinu zadaných požadavků a v některých částech zadání i přesahuje, ale na druhou stranu není tak dokonalá, aby se nedalo nic zlepšit. Jsou části, které nebyly dodělány z časových důvodů, u jiných jsem nepřišel na jejich kvalitní řešení apod. Existuje v aplikaci několik zdánlivě zbytečných součástí, které jsou však připraveny pro případné rozšíření. Jedná se většinou buď o části editoru, které jsem já sám nestihl dodělat, nebo o části potřebné pro rozšíření editoru o simulaci.

Začnu u částí potřebných pro rozšíření editoru. Jak je na první pohled patrné z vytvořeného diagramu např. na Obrázek 29, nad hranou je ikonka, ale žádný text. Zde má být zobrazena násobnost hrany a také její scénáře. Veškeré náležitosti jsou v objektovém i grafickém modelu

zahrnuty jak je vidět na Obrázek 31, ale nepracuje s nimi žádná metoda, která by jim přiřadila nějaké hodnoty.



Obrázek 31: BPmodel.gmfgraph – EdgeFigure

Další možnost rozšíření spočívá v duplikaci objektů. Toto je právě bod v zadání, který nebyl splněn. Téměř vše je připraveno ke zprovoznění duplikace, ale z časových důvodů k tomu nedošlo. Všechny elementy, u nichž je požadována tato funkcionalita, mají vlastnost „duplicated“, která jim zajistí vazby se všemi jejich duplikáty. Stačí pouze začít spravovat tuto vlastnost, čímž je myšleno vytváření a rušení duplikačních vazeb mezi duplikáty.

Ještě jedna nesplněná část zadání práce zbývá, a tou je možnost přidávání atributů objektům diagramu. Důvodem nesplnění této části zadání však nebyla časová tíseň, ale opomnění při analýze. Už tento fakt napovídá tomu, že přidat tuto funkcionalitu bude znamenat zásah do celého modelu, nejspíš i nové generování kódu.

Nyní se dostáváme k rozšíření aplikace o simulaci. Nebudu se zabývat tím, jak provádět simulaci nad výsledným diagramem, protože toto je předmětem 2. části diplomové práce zabývající se simulací, ale uvedu, jaké části editoru mají návaznost na simulaci.

Všechny elementy mají referenci na rodičovský element a také si uchovávají seznam hran, které je spojují s okolím. K tomu slouží atributy „assigned“ a „linked“. Tyto vlastnosti se jistě uplatní pří zpracování topologie vytvořeného diagramu, nad kterým se má provádět simulace.

Každé místo má atribut „contains“, který uchovává seznam objektů v něm obsažených. Při přesouvání objektů z jednoho místa do druhého tato vlastnost jistě najde své uplatnění.

4 Závěr

Dle zadání diplomové práce byla vytvořena funkční aplikace, která umožňuje vytváření Petriho sítí s možností tvorby podprocesů, editací parametrů jednotlivých elementů, jako jsou např. skripty, skupiny, scénáře apod. Aplikace by dále měla také zvládat vytváření více grafických reprezentací jednoho místa, ale této funkcionality nebylo dosaženo.

Já jsem však se svojí prací spokojený, jelikož z každého uplynulého projektu získám také mnoho zkušeností. V tomto případě jsem např. detailně pochopil tvorbu třídních diagramů, osvěžil jsem si znalosti javy a přiučil se opět nové technologie a v neposlední řadě jsem také pochopil důležitost verzovacího systému ve chvíli, kdy u mě došlo ke ztrátě veškerých dat i s diplomovou prací.

Tato diplomová práce se skládá za 2 samostatných diplomových prací, kde jednou je právě vytvoření editoru a druhou je simulace nad vytvořeným modelem. Editor je již hotový a připravený na další rozšíření, a to jak rozšíření samotného editoru, tak i případné simulace.

5 Literatura

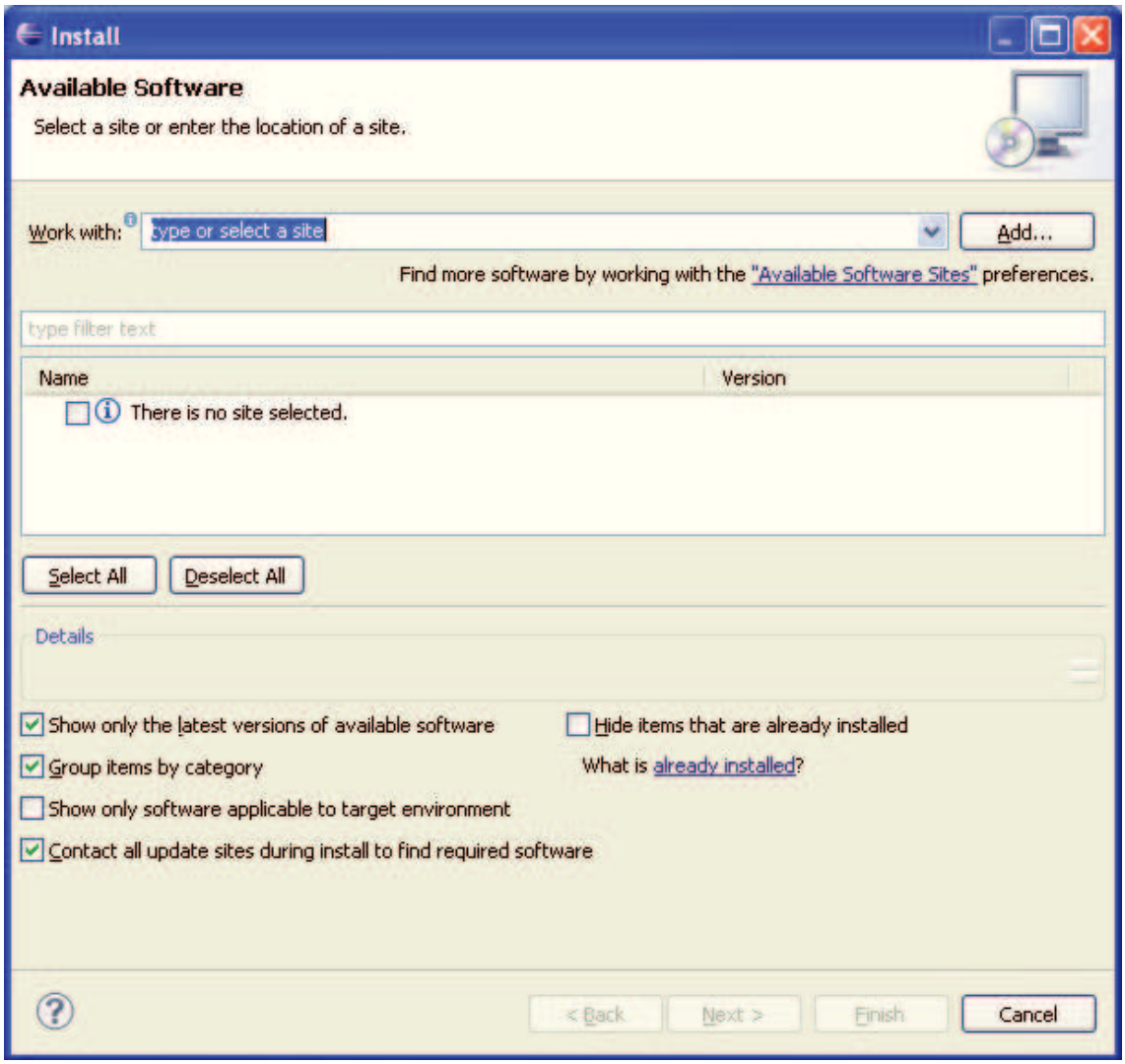
[1] ECLIPSE.ORG. *Graphical Modeling Framework* [online]. 2006-01-18, 2011-12-07 [cit. 2012-04-19]. Dostupné z: http://wiki.eclipse.org/Graphical_Modeling_Framework

[2] GRONBACK, Richard C. *Eclipse modeling project*: *a domain-specific language toolkit*. Upper Saddle River,: Addison-Wesley, c2009, 706 s. ISBN 978-0-321-53407-1.

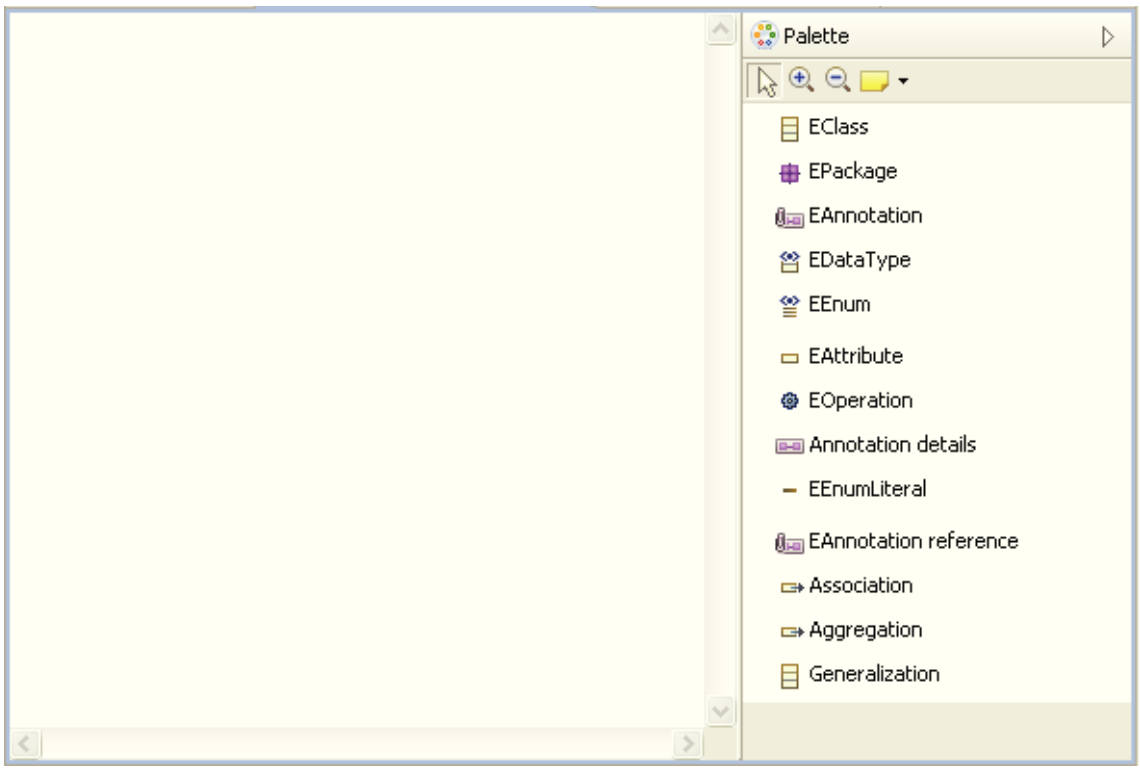
[3] BRAZEAU, Jean-François. *GMF Samples a tutorials overview* [online]. 2009-12-02 [cit. 2012-04-19]. Dostupné z: http://gmfsamples.tuxfamily.org/wiki/doku.php

[4] WRIGHT, Jevon. *GMF Diagram Partitioning* [online]. 2008-09-25 [cit. 2012-05-01]. Dostupné z: http://www.jevon.org/wiki/GMF_Diagram_Partitioning

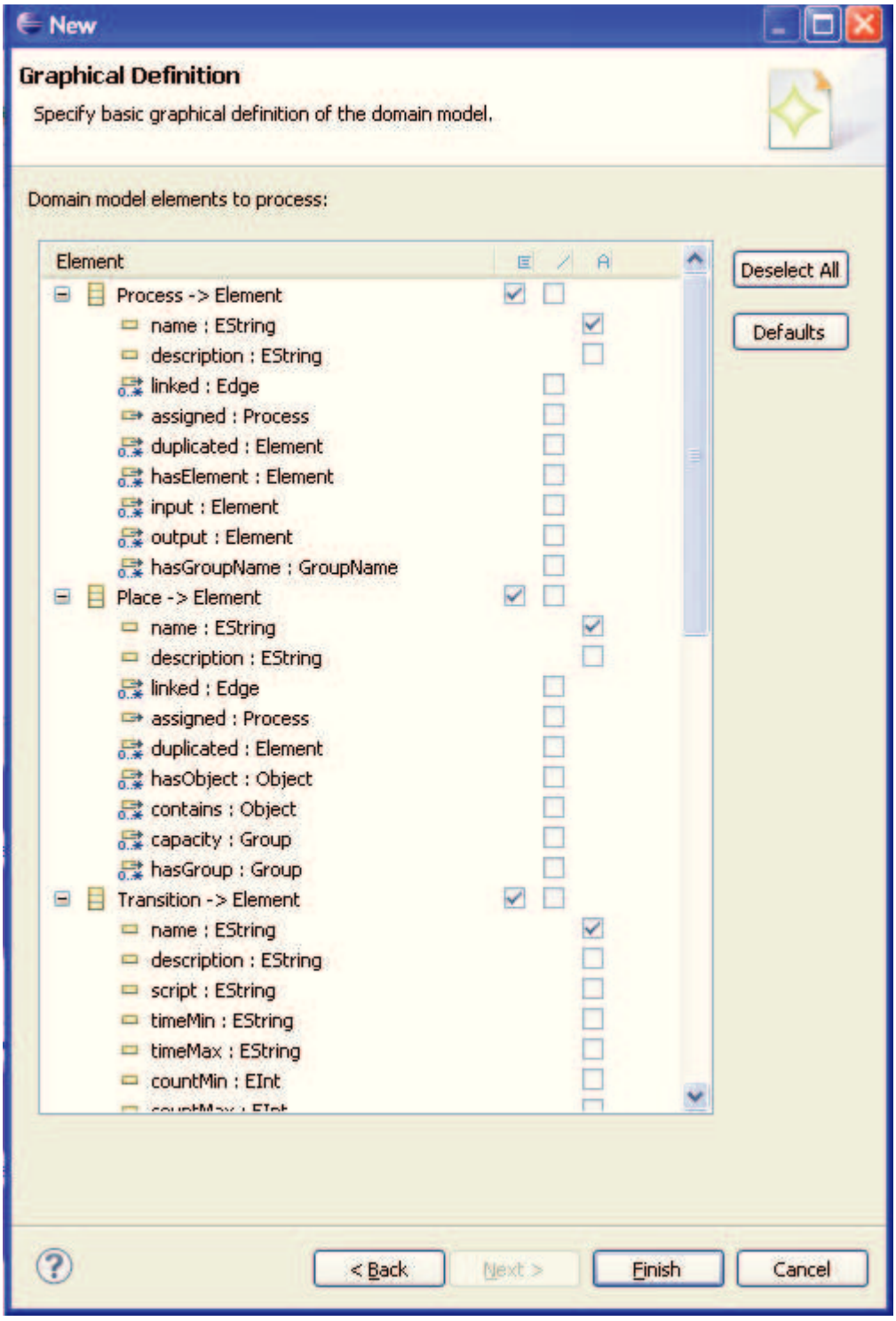
6 Přílohy



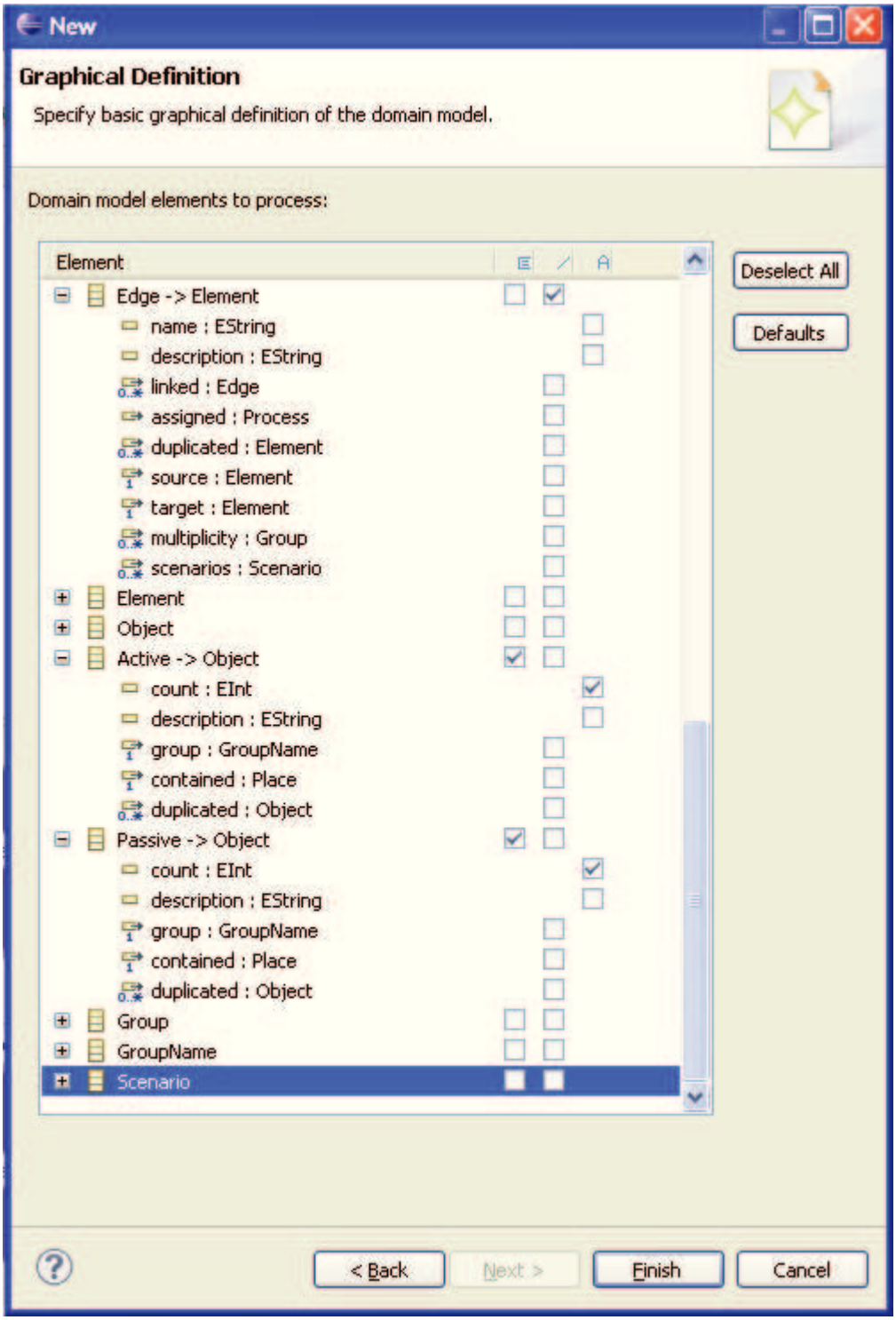
Obrázek 32: Okno Install



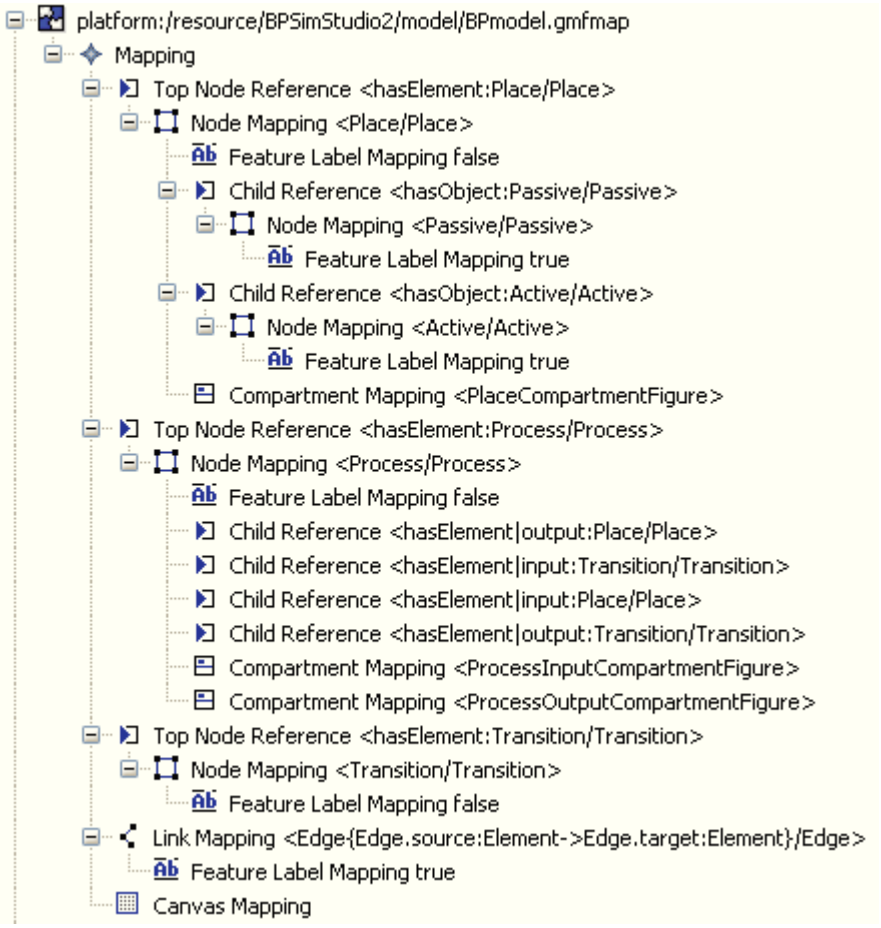
Obrázek 33: BPmodel.ecore_diagram



Obrázek 34: Graphical Definition (1/2)



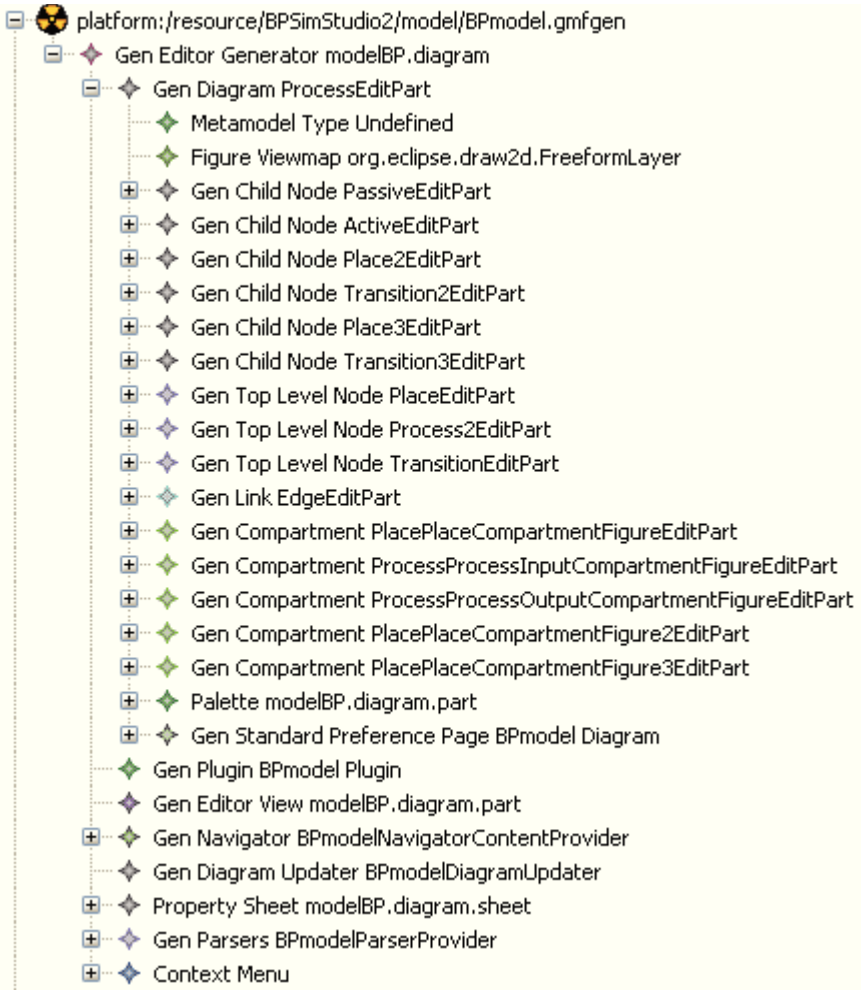
Obrázek 35: Graphical Definition (2/2)



Obrázek 36: BPmodel.gmfmap – upravený

```
public class CompartmentFigureListener implements FigureListener {

  private ListCompartmentEditPart compartmentEditPart = null;

  //předdefinovaná velikost elementu
  public static final int elementSize = 18;

  public CompartmentFigureListener(ListCompartmentEditPart
       compartmentEditPart) {
    this.compartmentEditPart = compartmentEditPart;
  }

  @Override
  public void figureMoved(IFigure f) {
    ResizableCompartmentFigure figure = (ResizableCompartmentFigure)f;
    if (figure.getSize().width != 0) {         //kontrola vykreslení
      //prvek compartment objektu figure
      IFigure contentPane = figure.getContentPane();
      //okraje objektu figure
      Insets is = figure.getInsets();
      //seznam elementů v compartment
      List<AbstractEditPart> childs =
           compartmentEditPart.getChildren();
      int count = 0;
      int resolution = elementSize + 2;
      for (AbstractEditPart child : childs) {
        if (child instanceof AbstractGraphicalEditPart) {
          //element
          AbstractGraphicalEditPart gEditPart =
               (AbstractGraphicalEditPart) child;
          int x = is.left;
          int y = count * resolution + is.top;
          Rectangle constraint =
               new Rectangle(x, y, elementSize, elementSize);
          //přiřazení vzhledu elementu
        contentPane.setConstraint(gEditPart.getFigure(),constraint);
          count++;
        }
      }
    }
  }
}
```

Výpis 7: CompartmentFigureListener



Obrázek 37: BPmodel.gmfgen